

NAVAL POSTGRADUATE SCHOOL MONTEREY, CALIFORNIA



THESIS

A LINUX-BASED APPROACH TO LOW-COST SUPPORT OF ACCESS CONTROL POLICIES

by

Paul C. Clark

September 1999

Thesis Advisor:
Second Advisor:

Cynthia E. Irvine
Dennis Volpano

Approved for public release; distribution is unlimited.

19991126 082

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 1999		3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE A LINUX-BASED APPROACH TO LOW-COST SUPPORT OF ACCESS CONTROL POLICIES			5. FUNDING NUMBERS	
6. AUTHOR(S) Clark, Paul C.				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) It is vital to our country's political and economic future to adequately protect corporate and government information from unauthorized disclosure and modification. Unfortunately, the current state of computer security is weak, especially when novice adversaries can perform successful infiltrations of sensitive systems. Systems that enforce Mandatory Access Control (MAC) policies are known to reduce some known security weaknesses, but such systems have seen limited use within the United States Government, and they are rarely applied in the private sector. Some of this limited use is caused by a lack of exposure to systems able to enforce MAC policies. This thesis presents an inexpensive approach to providing a system supporting MAC policies, allowing users an opportunity to have hands-on experience with such a system. A detailed design for modifying the Linux operating system is given, allowing for the flexible and simultaneous support of multiple policies. In particular, a design and detailed specification for the implementation of label-based interfaces for the mandatory portions of the Bell and LaPadula secrecy model and the Biba integrity model have been developed. Implementation of portions of this design has demonstrated the feasibility of this approach to label-based interfaces. This design has potential for widespread use in computer security education, as well as broad application as a component in the ongoing Department of Defense research of trusted computer system interfaces.				
14. SUBJECT TERMS Mandatory Access Control, Security Policy, Linux, Education			15. NUMBER OF PAGES 189	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

NSN 7540-01-280-5500
298 (Rev. 2-89)

Standard Form

Prescribed by ANSI Std. Z39-18 298-102

Approved for public release; distribution is unlimited.

**A LINUX-BASED APPROACH TO LOW-COST SUPPORT OF
ACCESS CONTROL POLICIES**

Paul C. Clark
B.S., California Polytechnic University, Pomona, 1990

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

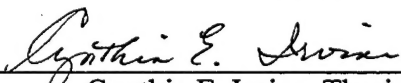
from the

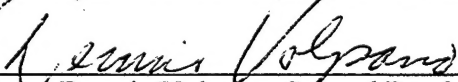
**NAVAL POSTGRADUATE SCHOOL
September 1999**

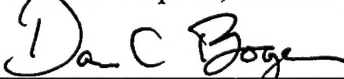
Author:


Paul C. Clark

Approved by:


Cynthia E. Irvine, Thesis Advisor


Dennis Volpano, Second Reader


Dan Boger, Chairman
Department of Computer Science

ABSTRACT

It is vital to our country's political and economic future to adequately protect corporate and government information from unauthorized disclosure and modification. Unfortunately, the current state of computer security is weak, especially when novice adversaries can perform successful infiltrations of sensitive systems. Systems that enforce Mandatory Access Control (MAC) policies are known to reduce some known security weaknesses, but such systems have seen limited use within the United States Government, and they are rarely applied in the private sector. Some of this limited use is caused by a lack of exposure to systems able to enforce MAC policies. This thesis presents an inexpensive approach to providing a system supporting MAC policies, allowing users an opportunity to have hands-on experience with such a system. A detailed design for modifying the Linux operating system is given, allowing for the flexible and simultaneous support of multiple policies. In particular, a design and detailed specification for the implementation of label-based interfaces for the mandatory portions of the Bell and LaPadula secrecy model and the Biba integrity model have been developed. Implementation of portions of this design has demonstrated the feasibility of this approach to label-based interfaces. This design has potential for widespread use in computer security education, as well as broad application as a component in the ongoing Department of Defense research of trusted computer system interfaces.

TABLE OF CONTENTS

I. INTRODUCTION	1
A. COMPUTER SECURITY AND THE OPERATING SYSTEM	1
B. OVERVIEW OF ACCESS CONTROL POLICIES.....	2
C. HISTORICAL BACKGROUND ON MAC SYSTEMS.....	4
D. ASSURANCE VERSUS POLICY	5
E. SECRECY AND INTEGRITY POLICIES AND MODELS	6
F. TRUSTED SUBJECTS VERSUS PRIVILEGED SUBJECTS	8
G. ADVANCING COMPUTER SECURITY	9
H. MAC SYSTEMS AT EDUCATIONAL INSTITUTIONS	10
I. SOLUTION REQUIREMENTS.....	11
1. Relatively Inexpensive.....	11
2. Runs on PC hardware.....	11
3. Dual-Boots with Other Operating Systems.....	11
4. Easy to Use	12
5. Supports Secrecy and Integrity Policies	12
6. Supports the Setting of a Session Level.....	12
J. PROBLEMS WITH EXISTING PRODUCTS.....	13
1. Microsoft Windows NT	13
2. Deep Purple.....	13
3. Sun Microsystems Trusted Solaris.....	14
4. Rule Set Based Access Control in Linux.....	14
5. Other Commercial Products Supporting MAC Policies	15
6. System Summary	16
K. PROPOSED SOLUTION	18
II. LINUX MANDATORY ACCESS CONTROL DESIGN	21
A. NEW DATABASES.....	21
1. Policy Label	21
2. Human-Readable Label (HRL) Databases.....	23
3. User Clearance Database	25
4. Range Database.....	26

B. NEW MODULES	26
1. Policy Modules	27
2. Meta-Policy Manager.....	27
3. Label Modules	27
4. Label Manager	28
5. Range Modules	28
6. Range Manager	28
7. Clearance Modules.....	28
8. Clearance Manager	29
C. LAYERING DESIGN	29
1. Clearance Manager Dependencies	30
2. Meta-Policy Manager Dependencies	31
3. Range Manager Dependencies.....	32
4. Label Manager Dependencies.....	33
5. Clearance Layer Dependencies.....	34
6. Range Layer Dependencies.....	35
7. Label Layer Dependencies.....	36
III. LINUX MODIFICATIONS.....	39
A. OPERATING SYSTEM MODIFICATIONS.....	39
1. Basic Policy Enforcement.....	39
2. Inode Changes.....	39
3. File Statistics.....	42
4. Subject Changes.....	42
5. Creating Objects.....	43
6. Deleting Objects.....	44
7. Deflection Directories	45
8. Updating Object Properties.....	46
9. The Super User	47
10. Setting Initial Policy Labels.....	48
11. Summary of Changes	49
B. APPLICATION MODIFICATIONS.....	49
1. File System Creation Program (mke2fs).....	49
2. Login Program (login)	50
3. Object Statistics (ls, stat).....	50
4. Process Status (ps)	51
5. Process Identification (id).....	51
6. Directory Creation (mkdir)	51
IV. CONCLUSIONS.....	53
A. PROGRESS MADE	53
B. PROBLEMS ENCOUNTERED	53

C. FUTURE RESEARCH	54
1. Additional User Roles	54
2. Auditing	55
3. Setuid and Setgid Programs	56
4. Deflection Directories	56
5. Administrative Interface	56
6. Privileges	57
7. Trusted Path	57
8. Policy Label Initialization	58
9. Move or Port to the Newest Linux Kernel	58
APPENDIX A. DESIGN DECISIONS RELATED TO COVERT CHANNELS	61
A. DEFLECTION DIRECTORIES	61
B. OBJECT PROPERTIES	62
APPENDIX B. DATABASE DESIGN	63
A. POLICY LABEL	63
B. LABEL DATABASE	64
C. USER CLEARANCE DATABASE	65
D. RANGE DATABASE	66
APPENDIX C. MODULE DESIGN	67
A. POLICY ENHANCED LINUX COMMON TYPES MODULE (PEL_TYP)	67
B. BELL AND LAPADULA POLICY MODULE (BLP_POL)	67
1. BlpPolInitLabel	68
2. BlpPolSetLevel	69
3. BlpPolGetLevel	70
4. BlpPolAddCategory	71
5. BlpPolDelCategory	73
6. BlpPolTestCategory	74
7. BlpPolDominates	75
8. BlpPolRead	77
9. BlpPolWrite	79
C. BIBA POLICY MODULE (BIB_POL)	81
1. BibPolInitLabel	82
2. BibPolSetLevel	82
3. BibPolGetLevel	83
4. BibPolAddCategory	85
5. BibPolDelCategory	86

6.	BibPolTestCategory	87
7.	BibPolDominates	89
8.	BibPolRead	91
9.	BibPolWrite	92
D.	BELL AND LAPADULA LABEL MODULE (BLP_LBL)	94
1.	BlpLblInit	96
2.	BlpLblBinToStr	98
3.	BlpLblStrToBin	100
E.	BIBA LABEL MODULE (BIB_LBL)	102
1.	BibLblInit	103
2.	BibLblBinToStr	106
3.	BibLblStrToBin	108
F.	BELL AND LAPADULA RANGE MODULE (BLP_RNG)	109
1.	BlpRngInit	110
2.	BlpRngSysLow	112
3.	BlpRngSysHigh	113
G.	BIBA RANGE MODULE (BIB_RNG)	114
1.	BibRngInit	115
2.	BibRngSysLow	117
3.	BibRngSysHigh	117
H.	BELL AND LAPADULA CLEARANCE MODULE (BLP_CLR)	118
1.	BlpClrGetClearance	119
I.	BIBA CLEARANCE MODULE (BIB_CLR)	122
1.	BibClrGetClearance	123
J.	LABEL MANAGER (LBL_MGR)	126
1.	LblMgrInitLabel	127
2.	LblMgrIsValid	128
3.	LblMgrBinToStr	129
4.	LblMgrStrToBin	131
5.	LblMgrGetBlp	134
6.	LblMgrGetBiba	135
7.	LblMgrSetBlp	136
8.	LblMgrSetBiba	137
K.	CLEARANCE MANAGER (CLR_MGR)	138
1.	ClrMgrGetClearance	139
L.	RANGE MANAGER (RNG_MGR)	141
1.	RngMgrGetRange	141
M.	META-POLICY MANAGER (POL_MGR)	143

1. PolMgrDominates	143
2. PolMgrRead	145
3. PolMgrWrite	146
APPENDIX D. SOURCE CODE	149
A. POLICY ENHANCED LINUX COMMON TYPES (PEL_TYP)	149
1. Peltyp.h	149
B. BELL AND LAPADULA POLICY (BLP_POL)	150
1. blppol.h	150
2. blppol_getset.c	151
3. blppol_access.c	156
C. LABEL MANAGER (LBL_MGR)	159
1. lblmgr.h	159
2. lblmgr_getset.c	160
D. META-POLICY MANAGER (POL_MGR)	164
1. polmgr.h	164
2. polmgr.c	165
LIST OF REFERENCES	169
INITIAL DISTRIBUTION LIST	173

ACKNOWLEDGEMENT

The author would like to acknowledge the financial support of the National Security Agency, Code R2, for allowing the purchase of the software and reference material used in this thesis. The work was performed under Contract H98230-R2-98-8004.

The author would also like to thank Prof. Irvine for the opportunity to work on this thesis topic, as well as her guidance, support and encouragement.

Last, but not least, my wife and family deserves the most gratitude of all for their support and patience during the performance of this investigation. Without their support, none of this would have been possible.

I. INTRODUCTION

A. COMPUTER SECURITY AND THE OPERATING SYSTEM

Computer systems have grown from the huge and costly mainframe environments of yesterday to the powerful distributed environments of today; from one large machine to a plethora of Personal Computers (PCs); from the relatively slow and featureless to the fast and user-friendly. Great strides in technology have taken place in a very short time, but despite these great advancements, computer security continues to be a major problem, as recognized at the highest levels of the United States Government [Refs. 1, 2].

In the early days of the mainframe computer, physical security was king; the system was housed in a large room behind locked doors where only authorized personnel were allowed access. One program ran at a time, submitted in batch mode in the form of punch cards, through a check-in procedure outside the computer room. The technicians in the room were given special trust to handle the cards and any resulting output with care, being discreet about any information they contained. The output and the original punch cards were returned to the submitting user through a checkout procedure outside the computer room. The submitter was required to keep the cards in a safe place and in their proper sequence. Most people never saw the expensive "monster" of a machine behind the white walls. This was arguably the best computer security the industry has ever achieved, but this accomplishment was not achieved through technology – it was achieved by implementing standard operating procedures as well as temporal and physical controls.

Computers now store programs on connected devices, each with many gigabytes of information. They run them "simultaneously," serving and being served by other computers connected in a vast network. If the network really is the computer, then the world is our computer room, giving everyone access to the power within its walls. Operating procedures and trusted computer technicians can no longer protect us from malicious programs and users, so we must now rely on the computer system to protect us

from each other. More correctly stated, the operating systems and network protocols are the only means of protection when computer systems are connected in a highly distributed fashion with a large number of users. This also applies to a closed network within an organization, where no access to the Internet is provided. Unfortunately, security is a secondary or tertiary concern to performance and functionality, and is often not even considered during the design process.

The first line of defense in today's computing environments is the operating system. This is the place where programs acting on behalf of users make requests for resources, such as files; this is where access to resources must be controlled. When a user requests a resource on the local system, the operating system should be able to determine whether the request should be granted or not, based on some kind of policy with respect to the user in question and the resource being requested. When a user requests a resource across a network, the operating system should be able to use network protocols to query the remote system whether the request should be granted or not.

B. OVERVIEW OF ACCESS CONTROL POLICIES

An information access control policy is a high-level description of how people can access information. An example is the government policy relating to sensitive information: one cannot access information that is classified higher than one's clearance. In other words, if a user is cleared to see information up to a SECRET level, then that person cannot see information classified at a TOP SECRET level. Another example is a policy that states employees in the accounting department cannot see files created by the engineering department, and vice versa. The operating system is the place where policies such as these are implemented and enforced.

When mapped to computers, policies are typically stated in terms of how "subjects" can access "objects." A subject can be thought of as the active entity in the system acting on behalf of a user, such as a process, task or thread. An object is a passive entity in the system, such as a file or directory.

Given any kind of policy, the elements of the policy can be factored into their basic properties, which in turn can be grouped into one of three categories: a Discretionary Access Control (DAC) policy, a Mandatory Access Control (MAC) policy, or a Supporting Policy. [Ref. 3, p. 55] A Supporting Policy is one that is used to support the proper enforcement of a DAC or MAC policy.

A DAC policy is one where an object (e.g., file or directory) has a named user, or set of users, who can decide, at their discretion, who can have access to the object [Ref. 4, p. 290]. This can be implemented in many different ways, with varying degrees of granularity. Many modern implementations include something known as an Access Control List (ACL), a feature that is available in Windows NT and some versions of Unix.

A MAC policy, on the other hand, implies that the owner of an object does not have control over who has access to it [Ref. 4, p. 290]. A good example is a file with a SECRET classification; the owner of the document does not have the discretion to control who has access to it because the policy states that a person must have the necessary clearance to view it. Systems that employ a MAC policy will often support a DAC policy as well, providing the ability to control access to objects within the same data classification. Putting a DAC policy "on top of" a MAC policy does not provide a significant increase in security, but it does provide end-users with the functionality they desire, and it may satisfy some set of functional security requirements, introduced in the next section.

Systems that enforce a DAC policy are easily obtained on the market, but it is much harder to find systems that enforce a MAC policy. The reasons for this disparity are not easily described, and tend to evoke strong disagreement between various camps of the government, research, and commercial sectors. The fact remains that there are few MAC systems, and they tend to be orders of magnitude more expensive than non-MAC systems.

C. HISTORICAL BACKGROUND ON MAC SYSTEMS

Systems that can withstand active and passive attacks from hostile users and outsiders have been research topics for decades. The Department of Defense (DoD) has been directly involved from the beginning, sponsoring numerous research projects. In fact, the 16 seminal papers on computer security were either sponsored by a DoD agency or written by an employee of the DoD [Ref. 5]. The DoD has also issued directives detailing minimum computer security requirements [Ref. 6].

The result of some of this activity was the establishment of the DoD Computer Security Center and the publication of the Trusted Computer System Evaluation Criteria (TCSEC) in 1983, commonly referred to as the "Orange Book" [Ref. 7], which was revised and republished in 1985. The Orange Book described the minimum requirements for various levels of assurance, and supported the evaluation of products that claimed to meet a level of assurance. The Orange Book was developed with the following three goals in mind:

...(a) to provide guidance to manufacturers as to what to build into their new, widely-available trusted commercial products in order to satisfy trust requirements for sensitive applications and as a standard for DoD evaluation thereof; (b) to provide users with a yardstick with which to assess the degree of trust that can be placed in computer systems for the secure processing of classified or other sensitive information; and (c) to provide a basis for specifying security requirements in acquisition specifications. [Ref. 7, p. v]

The Orange Book specifies four divisions of security assurance and functionality, ranging from the high end at division A to the low end at division D. Divisions B and C are broken down further, giving the following seven classes (from lowest to highest): D, C1, C2, B1, B2, B3, and A1. Class C1 and C2 have DAC requirements, while the higher classes have both DAC and MAC requirements (among other requirements).

Before the Orange Book was published, there was some research and development being conducted in the area of MAC systems, but it was not until after its publication that commercial companies became interested in meeting the stated needs and requirements of the DoD, as put forth in the Orange Book. Unfortunately, such ventures generally did not experience financial success over the next five to ten years, leaving some companies with the impression that MAC systems are not marketable. On the other hand, the impression left with the users was that MAC systems are expensive and unusable.

D. ASSURANCE VERSUS POLICY

A computer system can have security problems due to one or more of the following reasons:

- The enforced policy is flawed and it will never work.
- The policy was poorly implemented, either unintentionally (by a poor design) or maliciously (through poor configuration management).
- The final product is not configured and administered in a secure manner.

The quality of "assurance" relates to the first and second bullets listed above: 1) the measure of confidence that can be placed in the policy; and 2) the measure of confidence that can be placed in a computer that it correctly enforces its implemented policies. It is often wrongly assumed that if a computer implements a MAC policy, then it must be a very secure computer. In reality, the number of enforced policies and their restrictive properties has nothing to do with the measure of assurance. One can choose the most respected and strict policies ever conceived but implement them so poorly that the resulting system is very insecure and therefore has very low assurance. On the other hand, one can choose a flawed policy and implement it without any bugs and end up with a system that is of high assurance, because it implemented its policy well, even though the policy was not especially secure.

With respect to the expense of past and current MAC systems, the expense is a function of the level of assurance being engineered into the design and implementation, required by the security evaluation criteria, as well as the time and expense of the evaluation process. Often, assurance and MAC are incorrectly used as synonyms. This incorrect relationship may have started when the TCSEC increased functionality at each evaluation class as it increased assurance.

E. SECRECY AND INTEGRITY POLICIES AND MODELS

There are many MAC policies, but the two most common policies are some form of secrecy and/or integrity policies. A secrecy policy was described earlier, and is used by governments to classify documents and clear people; documents receive a classification such as SECRET, while a person receives a clearance such as TOP SECRET, allowing such a person to read anything that is classified from TOP SECRET and below. A secrecy policy is concerned with the controlled *disclosure* of information.

An integrity policy is very similar to a secrecy policy but is concerned with the controlled *modification* of information. As with secrecy, documents are given a classification, and people are given clearances. The kind of labels given to users and documents could be USER and ADMIN. A user with ADMIN integrity clearance can modify any document with the USER or ADMIN classification. A user with USER clearance can modify documents with the USER classification, but not with the ADMIN classification.

A security model is a simple, abstract, precise and unambiguous representation of a security policy [Ref. 3, p. 130]. Modeling a policy before it is implemented is one way of increasing the assurance of the implementation. For high assurance systems, the model is typically expressed in mathematical terms, allowing it to be subjected to mathematical proofs to expose any inconsistencies in the policy before it is implemented.

The two models often used as a basis for new secrecy and integrity models are the Bell and LaPadula model¹ and the Biba model, respectively [Ref. 4, p. 278].

The Bell and LaPadula (BLP) model has many properties² and functions seldom used in practice, such as its DAC functions [Ref. 3, p. 153]. However, the following parts are often used by systems that claim to support BLP:

- Simple Security Property

This is often referred to as the “read down” or “no read up” rule. It describes the basics of the secrecy policy. [Ref. 8, p. 16]

- Confinement Property or *-Property (pronounced “star-property”)

This is a property introduced to prevent accidental or malicious downgrading of information. It is often referred to as the “write up” or “no write down” rule. [Ref. 8, p. 17-18][Ref. 9, pp. 245-247]

- Compatibility Property

This property prevents the introduction of “dangling” objects by restricting the creation of objects in a hierarchy such that the secrecy level of the parent node (e.g., a directory) must be lower than or equal to the level of the child node. [Ref. 8, p. 29]

- Trusted Subjects

In practice, there must be some way to write information to lower levels under controlled situations. For example, as documents are declassified, how is a document read at a higher level and written to a lower level? To resolve this and other practical problems, a “trusted subject” is introduced. [Ref. 8, pp. 18, 64-67][Ref. 3, p. 153]

¹ Though it is often referred to as THE Bell and LaPadula model, the two authors actually published four distinct, yet related, models referred to as Volume I, Volume II, Volume III, and the Multics Interpretation.

² Historically, there is an inconsistent use of the terms “model”, “policy”, and “properties”. Bell and LaPadula actually restated the DoD security policy into what can be referred to as the Bell and LaPadula secrecy policy, consisting of additional security properties. The Bell and LaPadula model expressed the stated properties mathematically. However, the “Bell and LaPadula model” is often used synonymously with the “Bell and LaPadula policy.”

A trusted subject is allowed to read and write across a range of secrecy levels. Such a subject is still constrained by the simple security property because it is not able to read any higher than the upper end of its range. A trusted subject can write to objects within its range, but is unable to write lower than the lower end of its range, because of the confinement property.

For example, if a TOP SECRET file needs to be downgraded to SECRET, then a trusted subject with a range of SECRET to TOP SECRET is created. This range allows the subject to read the TOP SECRET file and write it to a SECRET file, but prevents it from writing the file to the UNCLASSIFIED level.

A system should have a very limited number of trusted subjects performing very specific tasks, because they have the obvious potential of creating a hole in a system's security. Such subjects should be subjected to extra scrutiny to avoid the insertion of malicious code.

The Biba model has the same properties as the BLP model, but with the following differences:

- Simple Security Property
This is often referred to as the "write down" or "no write up" rule.
- Confinement Property or *-Property
This is a property introduced to prevent the accidental or malicious corruption of data from low-integrity sources. It is often referred to as the "read up" or "no read down" rule.

F. TRUSTED SUBJECTS VERSUS PRIVILEGED SUBJECTS

There is a big difference between Trusted Subjects and Privileged Subjects. Trusted Subjects are part of the BLP and Biba models, and are still constrained by the simple security and confinement properties. On the other hand, privileged subjects are

those that are able to bypass security altogether. In either case, such subjects must be carefully introduced into a system because they have the potential of opening considerable security weaknesses. This may include the physical inspection of programming code, stringent configuration management procedures, or other means of providing some level of assurance that the subjects are not malicious.

G. ADVANCING COMPUTER SECURITY

It is vital to our country's political and economic future to adequately protect corporate and government information from unauthorized disclosure and modification. Unfortunately, the current state of computer security is weak, especially when novice adversaries can perform successful infiltrations of sensitive systems. Systems that enforce Mandatory Access Control (MAC) policies are known to reduce some of the security weaknesses, but such systems have seen limited use within the United States Government, and they have seen little or no use in the private sector. Some of this limited use is caused by a lack of exposure to systems able to enforce MAC policies, and the expense of current systems that do enforce a MAC policy.

Lack of exposure can be resolved through better education. There are probably many computer professionals who have never even heard of Mandatory Access Control. This is evidenced in the fact that few universities even offer an introductory course in computer security [Ref. 10], and in the cry for more education in the area of computer security in the Presidential Commission on Critical Infrastructure Protection (PCCIP) [Ref. 1, pp. 70-71]. Unlike most areas in the field of Computer Science, security runs across many disciplines. For example, databases and operating systems need security. More professors with expertise in these fields need to incorporate security into their courses. As students graduate from such programs and move into the work force they are more likely to consider product security requirements if they have some educational background in the area.

Lack of exposure could also be resolved if the cost of MAC systems was not so prohibitive. Ideally, every educational institution needing a MAC system to support their educational objectives could easily afford one, and every company that wanted to buy a MAC system could find one at the same price as a similar non-MAC system. In other words, the ideal situation would remove cost as a deciding factor.

H. MAC SYSTEMS AT EDUCATIONAL INSTITUTIONS

The Computer Security Track of the Computer Science Department of the Naval Postgraduate School (NPS) teaches a class called *Introduction to Computer Security*, among other computer security courses. This class is supported with a series of nine laboratory exercises, or tutorials, to enforce what the student is learning in class. Three of these tutorials are related to MAC policies. These three tutorials are critical in helping students fully understand these concepts.

The MAC tutorials are based on systems that are expensive to buy and maintain. This limits the track's ability to support the tutorials at distance learning sites, and prohibits other institutions from benefiting from our successes in this area because they are unable to buy the specialized hardware and/or software. Affordable MAC systems are absolutely necessary to provide the exposure and understanding of MAC systems within educational institutions.

On the other hand, lab space tends to be a premium commodity at educational institutions, so even if MAC systems were given away, they may not have any room for them. Therefore, everyone would benefit from a MAC system that does not require special hardware, and can run on computers that are commonly found in university computer labs. This is the case at NPS, where the Computer Security Lab desperately needs more space, but where the track must continue to maintain the specialized hardware and software.

I. SOLUTION REQUIREMENTS

This thesis researched the options for supporting an inexpensive Mandatory Access Control system. The requirements for such a system are given in the following subsections, along with a justification for each requirement.

1. Relatively Inexpensive

As expressed in previous sections the current high costs of MAC systems is a barrier to greater exposure, acceptance, and demand. MAC systems can help solve some of the security problems that currently plague DoD and non-DoD organizations. Some commands within the DoD recognize the need for MAC systems, but cost is not necessarily the obstacle. Conversely, the private sector does not currently recognize the need for MAC systems and therefore cannot justify their expense. Lowering the cost of MAC systems will increase their exposure, which will then increase their acceptance, which will cause consumers to demand such functionality from leading operating system vendors.

2. Runs on PC hardware

Requiring a MAC system to run on a PC may be viewed as a refinement of the previous requirement, though the previous requirement is directed at the cost of software; this requirement is directed at the cost of the hardware. By requiring the software to run on a PC, this eliminates the need for expensive and/or specialized hardware.

3. Dual-Boots with Other Operating Systems

This requirement is also cost-related. If, when a computer is booted, the user can choose between some number of operating systems to boot, the computer can have multiple uses. If the MAC system can be installed in a multi-boot configuration, then additional hardware and/or counter space is potentially not required to install and use it.

4. Easy to Use

As explained earlier, one of the complaints against MAC systems is their unusability. This has two perspectives: user and administrator. Both perspectives are equally important to address properly. Users cannot be expected to learn many new commands or other interfaces, so the MAC system interface must be fairly intuitive for new users. The administrative interfaces must be well documented with good security defaults, in case the documentation is not referred to. If an administrator is confused or frustrated with the system, then it will not be recommended for use.

5. Supports Secrecy and Integrity Policies

All current MAC systems support a secrecy policy because this policy is easily understood, both intellectually and in its application to an organization. Integrity, on the other hand, is not as easy to understand or apply, which may seem contrary to the previous requirement. From an educational point of view, however, a system supporting integrity is of great value, because it gives students hands-on experience with a somewhat difficult topic.

6. Supports the Setting of a Session Level

There are two basic types of user interfaces provided by MAC systems: 1) those that allow a user to read and write to any file as long as it falls within the user's clearance; and 2) those that require the user to specify the level at which reading and writing will be allowed. The setting of a level as described in the latter interface is known as setting a session level. Experience in the NPS Computer Security Lab has shown that the former interface is somewhat confusing to new users.

J. PROBLEMS WITH EXISTING PRODUCTS

This section lists systems that currently support at least one MAC policy, and describes why they do not meet the requirements set forth in the last section.

1. Microsoft Windows NT

Microsoft Windows NT server (version 4.0) costs about \$670 with 5 client licenses, while NT workstation costs about \$270 per license (without volume or other discounts). NT runs on PC hardware and can dual boot with many other operating systems, as long as it can read and write to the C partition. The user interface is very familiar to computer users because of its widespread use. However, effective administration of NT can be difficult to learn. NT supports a robust DAC policy, but it does not support a MAC policy.

2. Deep Purple

Deep Purple is the name of a product being sold by Argus Systems Group, Inc. It is a modified version of Microsoft Windows NT such that it supports a MAC secrecy policy, but no integrity policy. Argus Systems licensed the technology from the Defense Evaluation and Research Agency (DERA), an agency within the United Kingdom Ministry of Defense. DERA had performed a study, called Purple Penelope, that looked into the feasibility of modifying NT to support a MAC policy. Argus Systems took DERA's results and made the necessary modifications to make it commercially acceptable.

Deep Purple is an NT add-on product, meaning it is installed after NT and other user applications are installed. In addition to the cost of buying NT, a Deep Purple server license is \$795, while a Deep Purple workstation license is \$395. There are some additional administrative tasks, but it comes with good documentation. It does not support a session level.

3. Sun Microsystems Trusted Solaris

Sun Microsystems has a MAC-based version of its Solaris, Unix-based operating system called Trusted Solaris, supported since 1994. The NPS Computer Security Lab has been using version 1.1 of Trusted Solaris since 1995. The current version of Trusted Solaris is 2.5, with a list price of \$895 per workstation license, and \$4,995 per server license, before any academic discounts.

Trusted Solaris requires the Sun SPARC hardware, unlike the non-MAC Solaris operating system, which can also run on a PC. In addition, it only supports a secrecy policy. The user interface to Trusted Solaris is almost identical to the non-MAC version, and is easy for Unix users to adapt to. Administration of the system is more difficult for Unix administrators to adapt to, but the documentation is good. It does not support a session level.

4. Rule Set Based Access Control in Linux

Rule Set Based Access Control (RSBAC) runs on Linux, a variant of the Unix Operating System. Linux has been developed by many volunteers around the world and can be downloaded from many different sites at no charge, or purchased in an integrated package from various vendors at a very modest price, ranging from \$50 to \$150. This price is a one-time cost for an unlimited-use license. The source code is also freely available for users to examine and modify with a few restrictions aimed at making sure the resulting modifications are also freely available. As with most other variations of Unix, Linux only supports a DAC policy.

A German graduate student modified Linux as part of his thesis work to support a variety of MAC policies, which he dubbed "Rule Set Based Access Control" (RSBAC) [Ref. 11]. He provides the source code for his changes, and updates the code as he adds more functionality, and as new versions of Linux become available. Because of its Linux

roots, it can dual-boot with other operating systems and runs on a wide range of platforms, including the PC. It does not support a session level.

From a Linux user's point of view, there is little difference between RSBAC and Linux. However, from an administrator's point of view, it is frustrating to install and configure due to the poor documentation, which is nearly non-existent. The robustness of the implementation is also questionable. While conducting tests, an installation was quickly and unintentionally corrupted beyond repair.

5. Other Commercial Products Supporting MAC Policies

The only other commercially available operating system products that support a MAC policy are the XTS-300 from Wang Federal, Inc., and the ACF2 MVS with ACF2 MAC from Computer Associates International. All other Operating Systems supporting a MAC policy were produced prior to 1996, making their availability somewhat questionable.

Even though the XTS-300 runs on a PC, the initial purchase of an XTS-300 can be an order of magnitude greater than an ordinary PC. It is also questionable whether it can dual-boot with other operating systems. The cost of annual hardware and software maintenance for two XTS-300's exceeds that of six PC's. The product from Computer Associates (CA) runs on the J and 9000T models of the IBM ES/3090 series hardware. This is a mainframe product, the price of which would be prohibitive for academic users.

The XTS-300 supports both secrecy and an integrity policy, while the CA product only supports a secrecy policy. The documentation for the XTS-300 is extensive, while the quality of the CA documentation is unknown. The XTS-300 supports a session level.

6. System Summary

Table 1 provides a summary of the products described in this section and how they relate to the requirements set forth in the previous section. Because cost is such a relative measurement, actual costs are supplied in the table instead of a "yes" or "no" response.

Product	Software Cost Client	Software Cost Server	Runs on PC	Dual Boot	Easy to Use	Secrecy Policy	Integrity Policy	Session Level
Windows NT	\$270	\$670	Yes	Yes	Yes	No	No	No
Deep Purple ¹	\$665	\$1,465	Yes	Yes	Yes	Yes	No	No
Trusted Solaris	\$895	\$4,995	No	No	Yes	Yes	No	No
RSBAC	\$0	\$0	Yes	Yes	No	Yes	No	No
XTS-300 ²	\$25,000	\$25,000	Yes	No	No	Yes	Yes	Yes
ACF2 MVS	\$\$\$	\$\$\$\$	No	No	?	Yes	No	Yes

Table 1. Product-to-Requirements Summary

¹ The cost of NT is included.

² The cost is taken from a quote provided in 1997.

K. PROPOSED SOLUTION

There is no product that meets the stated requirements, as shown in Table 1. Building an operating system from scratch to meet the requirements would be an enormous undertaking, beyond the abilities of most people (and companies) to produce. The proposed solution is to modify an existing operating system. This is the path of least resistance, and the one most likely to succeed in a short amount of time. The Linux operating system is the best choice because it meets all the requirements except those related to MAC, although some would argue that it is not user friendly.

The remainder of this thesis describes modifications made to the Linux operating system to support additional access control policies. This effort started as an attempt to support secrecy and integrity policies, but it soon became obvious that a flexible design would allow any new policy to be easily added to Linux. In this thesis, the final design is referred to as "Policy-Enhanced Linux".

This effort does not attempt to provide a high-assurance product. As explained earlier, there is no connection between MAC policies and assurance. Policy-Enhanced Linux is meant to be used for educational purposes only, and should not be used to support "live" environments, whether within the DoD or without.

The remainder of this thesis is organized as follows:

- Chapter II gives a high-level design of the new databases and modules needed to support additional policies in Linux.
- Chapter III gives a high-level description of the changes required in the existing Linux source code to support new policies.
- Chapter IV provides a summary of the thesis work, including implementation progress, problems encountered, and suggested future research topics.
- Appendix A provides justification for some of the design choices.
- Appendix B provides detailed design specifications for new configuration databases.

- Appendix C provides detailed code specifications for new modules.
- Appendix D provides a listing of the implemented source code.

THIS PAGE INTENTIONALLY LEFT BLANK

II. LINUX MANDATORY ACCESS CONTROL DESIGN

This chapter describes the design of the new policy interface with the goal of providing a design that is independent of the operating system used to implement it. Chapter III continues by providing a design that incorporates these ideas into a Linux distribution. The designs presented in this chapter are intended to provide a somewhat generic interface to allow different policies to be “plugged into” an installation of Policy Enhanced Linux, thus making a change in policy relatively easy to achieve.

A. NEW DATABASES

This section describes the new databases that must be added to Linux to support a new policy. In this context the term “database” refers to a passive object holding information.

Each new database has a field to track its version. This provides a mechanism for a module that manages a database to be able to identify older versions of a database during run-time, and to act accordingly.

All the databases, with the exception of the Policy Label, will be stored in separate text files. This approach follows the standard Unix practice of having human-readable configuration files that can be modified using a text editor. It also has the added benefit of not requiring special administrative commands to be implemented. Such interfaces can be added later as a convenience for the System Administrator, but are not necessary for an initial implementation.

1. Policy Label

To support any kind of MAC policy, there must be a way of “attaching” some kind of label that describes the mandatory permission properties of the subject or object it is associated with. In a secrecy policy, for example, there must be a way to associate

labels such as "Secret" and "Top Secret" with files. An efficient way to implement such a label is with a set of bits, instead of a character string like "Top Secret". The Policy Label is a set of bits that becomes an immutable¹ property of each subject and object. By comparing the label of the subject with the label of the object, the system can easily determine whether the desired access should be allowed or not. The label also has a version number.

To create a flexible policy environment, the Policy Label is viewed as a container for several labels. In this initial design, the Policy Label will contain a secrecy label component and integrity label component, as shown in Figure 1.

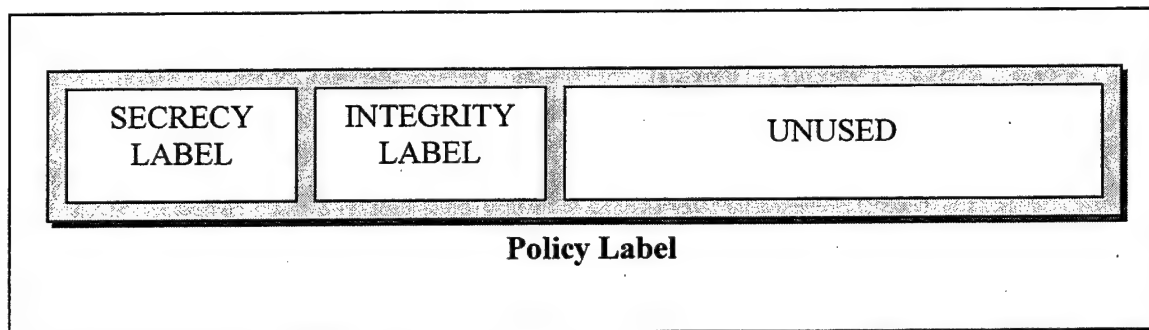


Figure 1. High-Level View of the Policy Label

Figure 1 shows a Policy Label with the labels of two enforced policies: secrecy and integrity. To add another enforced policy, the first step is to reserve some piece of the unused portion of the Policy Label for its use. Note that the label is not called an "Access Class" or "Access Label" due to the fact that the implementation is so generic that the label could be used to enforce a policy that is not related to access control, e.g., an Audit Policy.

¹ Immutability is certainly the desired property for labels, but no extra effort is expended in this design, above and beyond what Linux already provides, to guarantee that a policy label cannot be modified.

2. Human-Readable Label (HRL) Databases

Text-based labels are not actually associated with subjects and objects, rather, the Policy Label is a string of bits that represents a label that humans find easier to read and understand. For example, a simplistic implementation would associate the number zero with a human-readable label of "Unclassified" and a number three with a label of "Top Secret." There must therefore be a way of mapping the binary internal representations of a label to its human-readable form, and vice versa. The Human-Readable Label Databases help to satisfy this requirement, as shown in Figure 2.

<u>Bits in a Policy Label</u>	<u>Human-Readable Label</u>		
<table border="1"><tr><td>0</td><td>0</td></tr></table>	0	0	UNCLASSIFIED
0	0		
<table border="1"><tr><td>0</td><td>1</td></tr></table>	0	1	CONFIDENTIAL
0	1		
<table border="1"><tr><td>1</td><td>0</td></tr></table>	1	0	SECRET
1	0		
<table border="1"><tr><td>1</td><td>1</td></tr></table>	1	1	TOP SECRET
1	1		

Figure 2. Mapping Bits to Human-Readable Military Policy Labels

In order to support the addition of policies to the system, there is a separate human-readable label database for each enforced policy. Because the initial implementation will support the Bell and LaPadula secrecy policy and the Biba integrity policy, two Human-Readable Label Databases are required, as graphically shown in Figure 3.

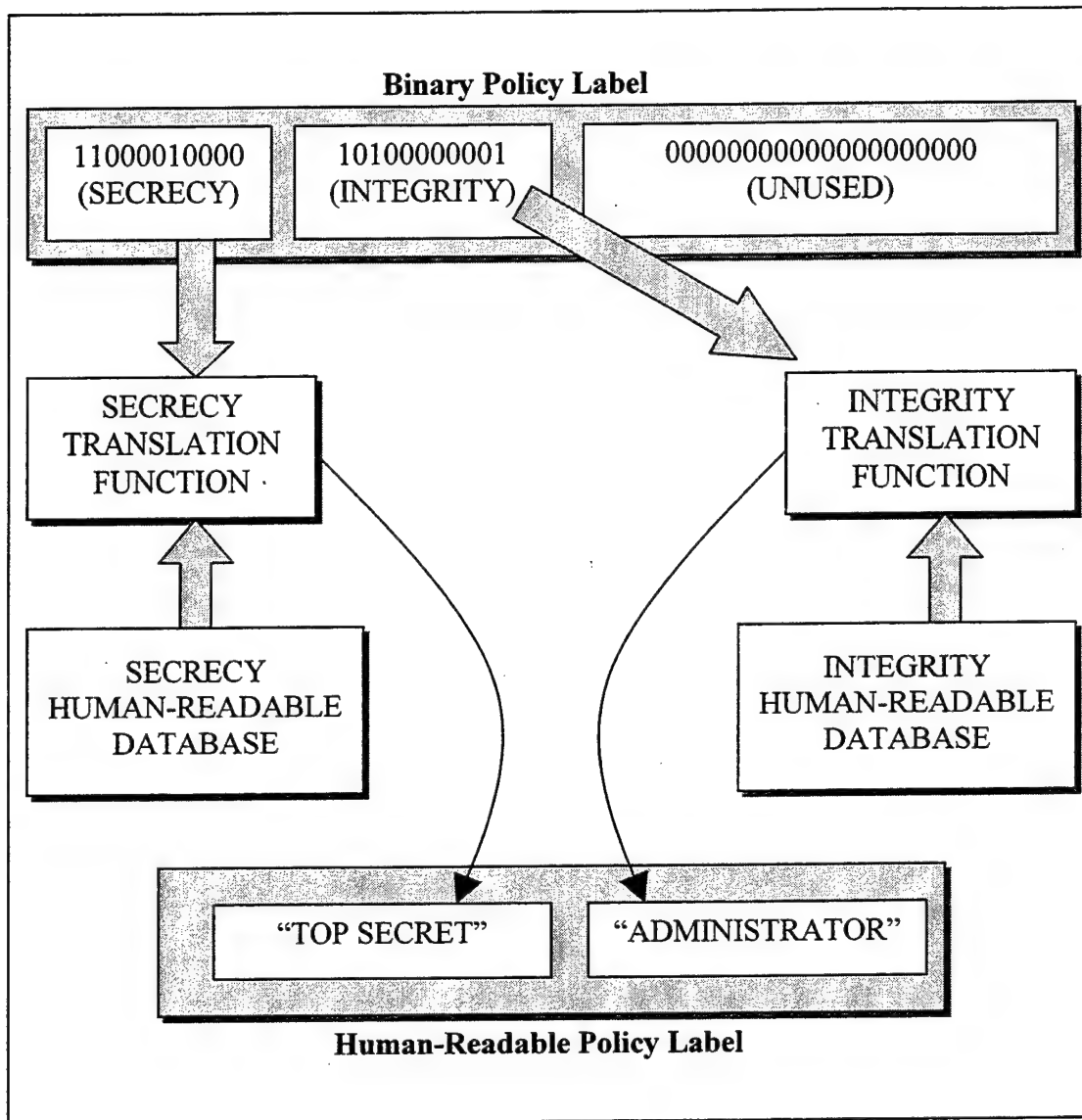


Figure 3. Binary to Human-Readable Label Translation

When a new policy is added to the system, an associated human-readable label database and translation function must be added. This is an important feature because it allows the addition of new policies while minimizing the changes to existing code.

Each database consists of two sections, as described below:

- A version number.
- A mapping of the possible bit values to human-readable form. The actual format of this section is policy-dependent and cannot be specified in advance.

3. User Clearance Database

To support a mandatory policy, there must be a way of associating a clearance with each user. For example, user A may have a clearance to only read objects that are less than or equal to the "Secret" classification, while user B may read any object up to "Top Secret". These settings need to be stored in a database separate from the usual Unix user attribute file (/etc/passwd) for compatibility purposes. This database is known as the Clearance Database.

Once again, in order to provide for flexibility, there is a separate database for each enforced policy. This database must hold the following information for each authorized user of the system:

- Version Number
- Minimum Session Level

This is the lowest level that a user can set for a session level. A user can still read data that this level dominates but this level becomes the lowest level where files can be created or modified by the user.

- Clearance

This is the highest level that a user can set for a session level. This becomes the highest level that a user can read or write files.

- Default Session Level

This is the session level that is set for the user if none is specified at login time, or for pseudo-users that are used for various system daemons that are started automatically by the system.

The following must be true:

$$\text{Minimum Session Level} \leq \text{Default Session Level} \leq \text{Clearance}$$

4. Range Database

To be able to bound the level of the data being produced and the level of the subjects being executed (despite what the user clearances are), there exists a database known as the "Range Database." This database allows a System Administrator to constrain the range at which a system will operate. For example, assume a user has a clearance ranging from UNCLASS to TOP SECRET on a system that is intended to only store information from CONFIDENTIAL to SECRET. The Range Database is used to set the range of data that can be created on a system. This range can change over the lifetime of a system so that there can exist objects on a system that are outside of the currently set range. There exists one database for each enforced policy, storing three pieces of information:

- The Version Number
- The system high label for the associated policy
- The system low label for the associated policy

The Following must be true:

$$\text{System low label} \leq \text{System high label}$$

B. NEW MODULES

This section describes the new modules to be added to Linux to support flexible policies. In this context, the term "module" refers to an active part of the system that manages a particular database or flow of control.

1. Policy Modules

For each enforced policy there must exist a module which enforces the policy. These modules provide an interface, as described below, with respect to the individual policy:

- Determine whether one label dominates another.
- Determine whether a read or write access (or read/write access) should be allowed based on the subject and object labels involved.
- Change or query attributes of a label.
- Publish properties of the policy that are needed by other modules, e.g., the number of secrecy levels supported by the secrecy policy implementation.

2. Meta-Policy Manager

The Meta-Policy Manager is a replaceable module that is responsible for calling the individual policy modules and returning the net result of the query. It is called the "Meta-Policy Manager" because it implements a policy on policies, deciding which policy modules are called first and whether some combination of results can result in an approved or declined access. It is expected that in the majority of cases, if not all cases, a positive response must be returned from all enforced policies in order to obtain the desired access. However, there is enough flexibility that some unforeseen set of policies could in fact support a situation where a negative result from one of the enforced policies could be over-ridden by a positive result from some other set of policies.

3. Label Modules

For each supported policy there exists a module that manages its associated Label Database. It provides an interface as described below:

- Map a human-readable label for the policy to a binary label for the policy.
- Map a binary label for the policy to a human-readable label for the policy.

4. Label Manager

The Label Manager is a replaceable module that defines the Policy Label, and is responsible for calling the individual Label modules to provide an interface as described below:

- Map a human-readable label for the system to a Policy Label for the system.
- Map a binary Policy Label for the system to a human-readable label for the system.
- Extract binary policy data from a Policy Label for any of the policies represented in the label.
- Set binary policy data in a Policy Label for any of the enforced policies.

5. Range Modules

For each enforced policy there must exist a module which manages its associated range database. Each range module provides an interface to do the following:

- Return the system low label for the associated policy.
- Return the system high label for the associated policy.

6. Range Manager

The Range Manager is a replaceable module that is responsible for calling the individual Range Modules to provide an interface for doing the following:

- Return the combined system low label for all the enforced policies.
- Return the combined system high label for all the enforced policies.

7. Clearance Modules

For each enforced policy there exists a module which manages the associated User Clearance Database. It provides an interface to do the following:

- Return the maximum clearance for a given user ID
- Return the minimum clearance for a given user ID
- Return the default session level for a given user ID

8. Clearance Manager

The Clearance Manager is a replaceable module that is responsible for calling the individual Clearance Modules to provide an interface to do the following:

- Return the maximum session level allowed for a given user, in the form of a complete Policy Label.
- Return the minimum session level (clearance) for a given user, in the form of a complete Policy Label.
- Return the default session level for a given user., in the form of a complete

C. LAYERING DESIGN

This section describes how the modules are layered. The design is such that the higher layers are dependent on the lower layers in a loop-free construct; modules in a particular layer do not call modules in the same or higher layer. This section includes the names of the modules enforcing the initial policy, whereas the last section described the design in a very generic manner. The layering design is shown in Table 2. Note that “BLP” is short for “Bell and La-Padula”.

Layer Name	Module Name		
Control Layer	Clearance	Range	Meta-Policy
	Manager	Manager	Manager
Label Utility Layer	Label Manager		
Clearance Layer	BLP Clearance	Biba Clearance	
Range Layer	BLP Range	Biba Range	
Label Layer	BLP Labels	Biba Labels	
Policy Layer	BLP Policy	Biba Policy	

Table 2. Policy Enforcement Layering Design

The following subsections describe the dependencies of the modules. In the figures provided in these subsections, modules with an arrow pointing to another module indicate dependencies.

1. Clearance Manager Dependencies

The Clearance Manager depends on the following layers: Label Utility Layer, Clearance Layer, and the Policy Layer. The Clearance Layer provides the clearances for a given user in binary format. It then assembles a complete Policy Label that represents the overall clearance of the user by calling the Label Manager. Figure 4 graphically illustrates the dependencies.

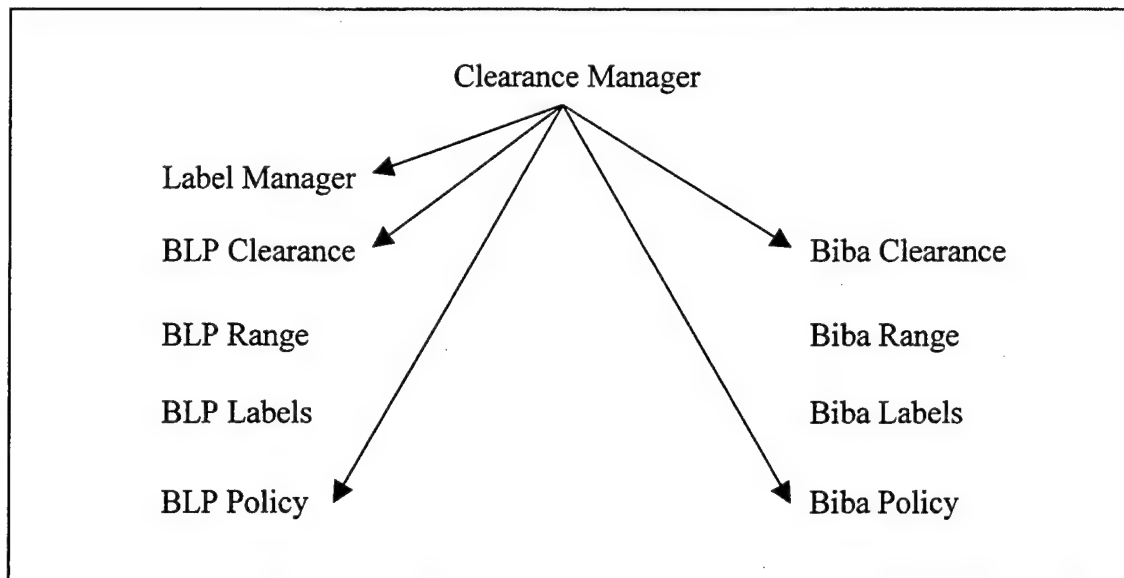


Figure 4. Clearance Manager Dependencies

2. Meta-Policy Manager Dependencies

The Meta-Policy Manager depends on modules in the Label Utility Layer and the Policy Layer. It uses the Label Manager to extract individual pieces of a Policy Label corresponding to the enforced policies of the system. These components are then passed to the respective modules in the Policy Layer to determine the dominance relationship between two Policy Labels. Figure 5 graphically illustrates the dependencies.

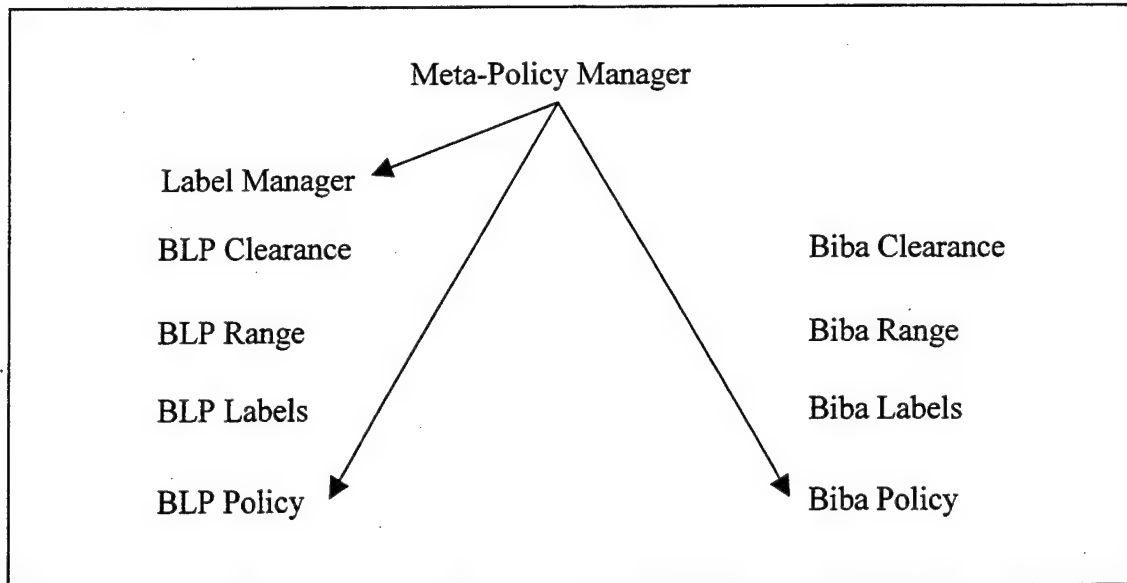


Figure 5. Meta-Policy Manager Dependencies

3. Range Manager Dependencies

The Range Manager depends on the modules in the following layers: Label Utility Layer, Clearance Layer and the Policy Layer. It uses information from the Range Layer and Policy Layer to obtain the currently configured system high and system low labels for each enforced policy in binary format. It then uses the Label Layer to combine the individual policy labels into one combined system high and system low Policy Label. Figure 6 graphically illustrates the dependencies.

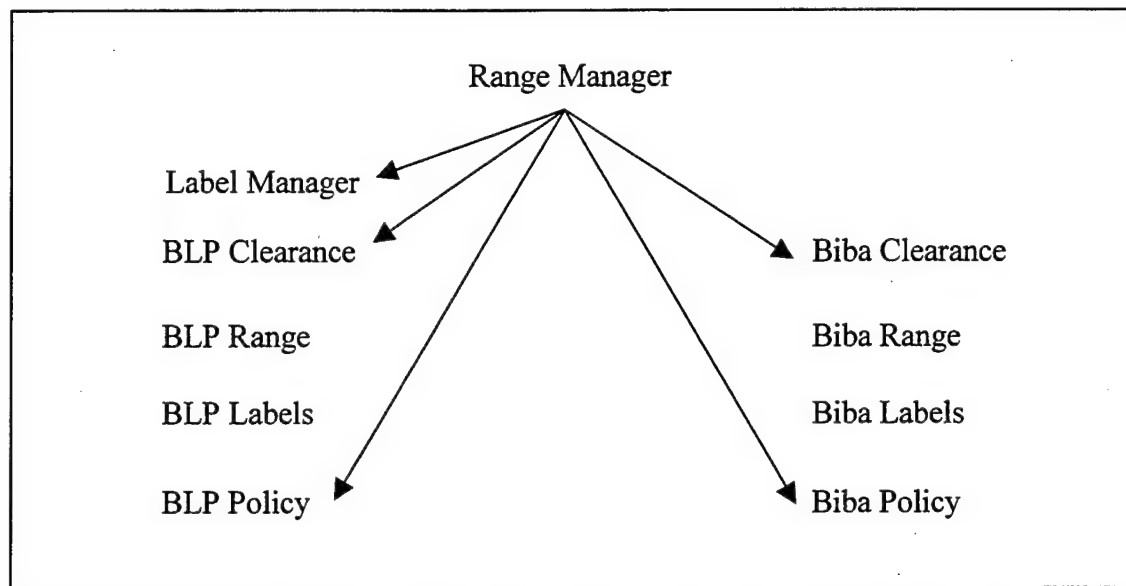


Figure 6. Range Manager Dependencies

4. Label Manager Dependencies

The Label Manager depends on the modules in the Label Layer and the Policy Layer. Given a binary Policy Label, the Label Manager can convert it to a Human-Readable Label with the information provided by the Label Layer. In addition, given a Human-Readable Label, it can convert it to a binary Policy Label with information provided by the Label Layer. Figure 7 graphically illustrates the dependencies.

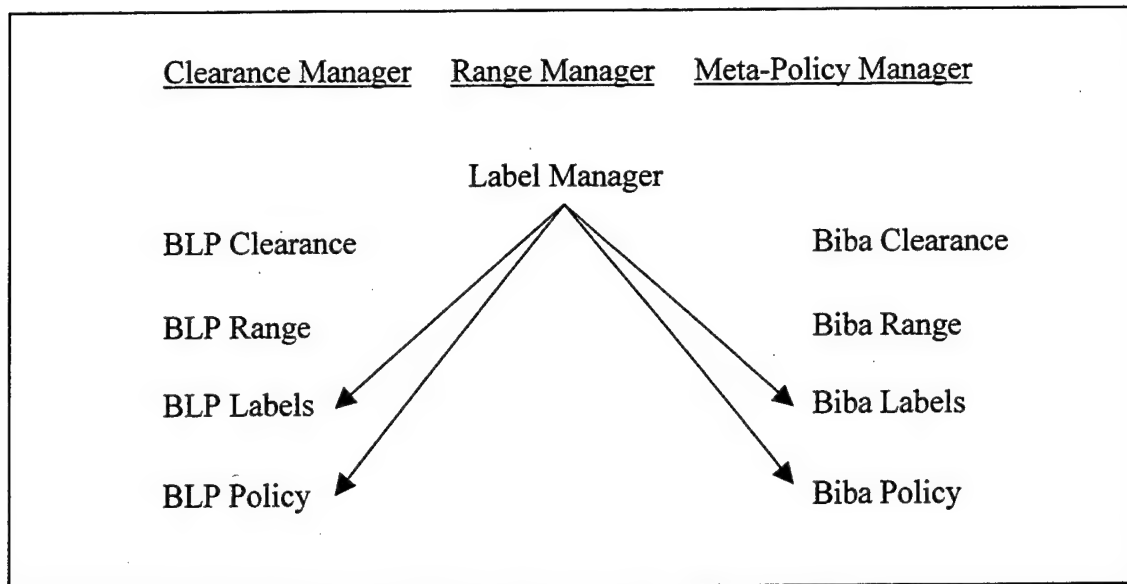


Figure 7. Label Manager Dependencies

5. Clearance Layer Dependencies

The various Clearance modules within the Clearance Layer are all dependent on modules that exist in both the Label and Policy layers to provide label definitions and translations. For example, the BLP Range module requires the services of the BLP Label module to translate the configured system low and system high secrecy value from human-readable form to binary form. The BLP Policy module defines the BLP secrecy label. Figure 8 graphically illustrates the dependencies.

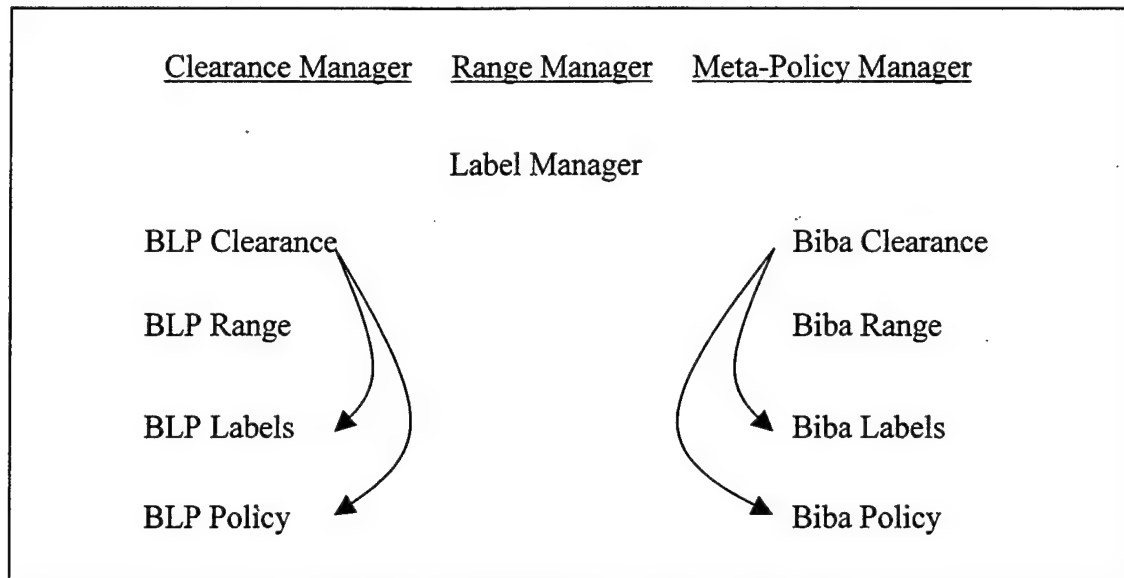


Figure 8. Clearance Layer Dependencies

6. Range Layer Dependencies

The various Range modules within the Range Layer are all dependent on modules that exist in both the Label and Policy layers, to provide label definitions and translations. For example, the BLP Range module requires the services of the BLP Label module to translate the configured system low and system high secrecy value from human-readable form to binary form. The BLP Policy module defines the BLP secrecy label. Figure 9 graphically illustrates the dependencies.

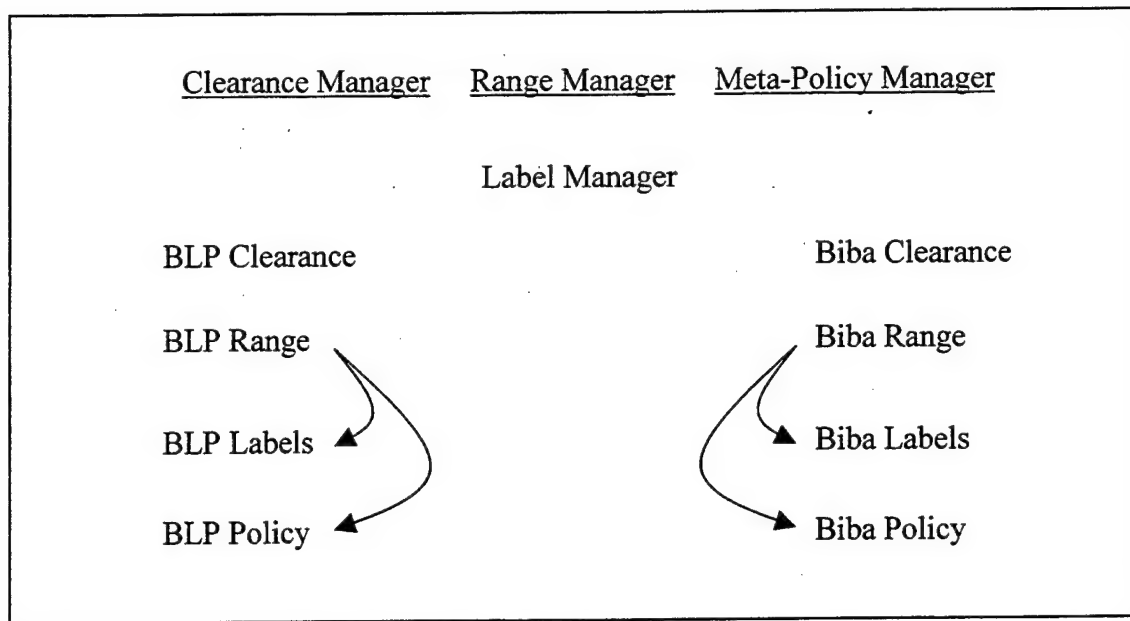


Figure 9. Range Layer Dependencies

7. Label Layer Dependencies

The various Label Modules within the Label Layer are all dependent on the associated policy modules of the Policy Layer to provide the maximum acceptable values for the policy-dependent binary labels. For example, the BLP Policy module must provide the maximum number of secrecy levels allowed given the BLP label defined by that module. Figure 10 graphically illustrates the dependencies.

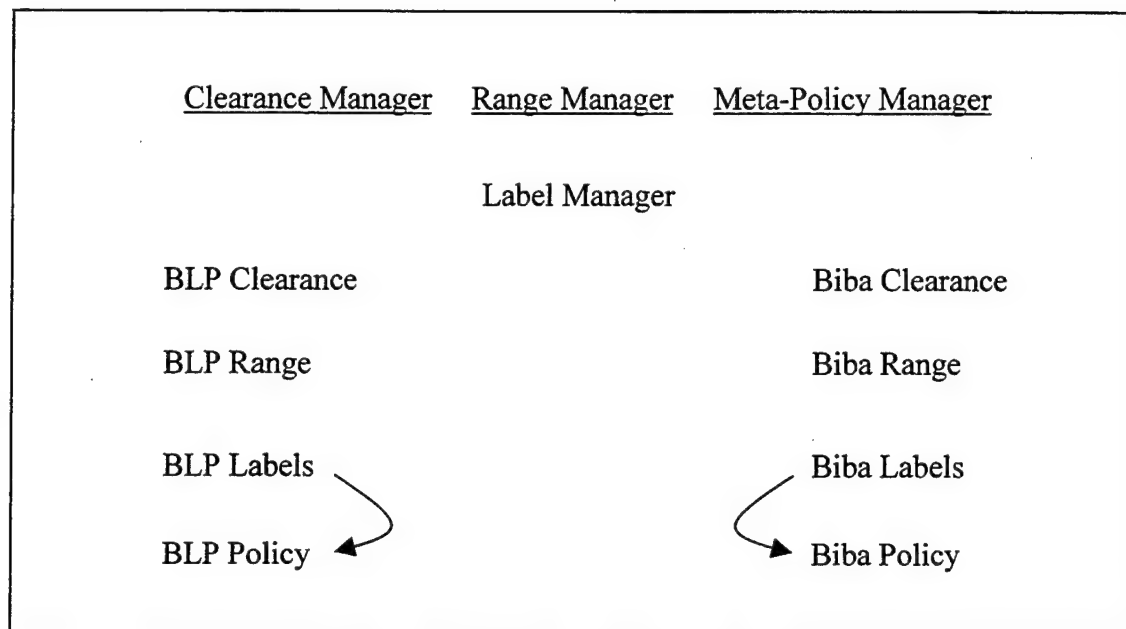


Figure 10. Label Layer Dependencies

THIS PAGE INTENTIONALLY LEFT BLANK

III. LINUX MODIFICATIONS

This chapter is divided into two major parts to describe the changes that need to be made to Linux to support the designs proposed in Chapter II. These two sections are “Operating System Modifications” and “Application Modifications”.

A. OPERATING SYSTEM MODIFICATIONS

1. Basic Policy Enforcement

At the core of every access control policy is a description of how subjects can read or write objects. Therefore, any system call that checks the kind of access given to a subject must be modified to call the Control Layer to perform the additional policy checks. The following system calls are affected:

- open
- opendir
- access

Other system calls are not affected, such as `read()` and `write()`, because the `open()` call determines the permission that is given to the opening subject when the file is opened.

2. Inode Changes

Linux is designed to simultaneously provide up to 15 different kinds of file systems. This is done by providing a common file system interface, no matter which file system is being accessed, which in turn calls the file system-specific interface to take the appropriate action on a file system object, such as a file or directory. In Linux terms, the common file system interface is referred to as the Virtual File System (VFS), whose structures only exist in memory during execution, going away when the system is shut down.

The VFS relies on the underlying persistent file system data structures stored on disk to keep track of the visible objects and to provide the interface to perform operations on those objects. The native Linux file system is known as the Second Extended File System (EXT2) and has been available since 1993. It is this file system which has served as the starting point for the design of Policy Enhanced Linux, viz., the EXT2 file system has been modified to provide MAC. The VFS also had to be modified in order to provide the necessary functionality.

Every object in the Linux file system has a unique structure associated with it, called an inode, which keeps track of various properties of the object, e.g., its owner, creation time and access rights, as well as the location on the disk where the object is being stored. The inode is the obvious place to store the Policy Label for file system objects. Therefore, all objects managed by Linux via inodes will be subjected to the new policies. Figure 11 graphically shows an inode structure with some of its data elements, and a representation of how its associated file is linked to disk blocks.

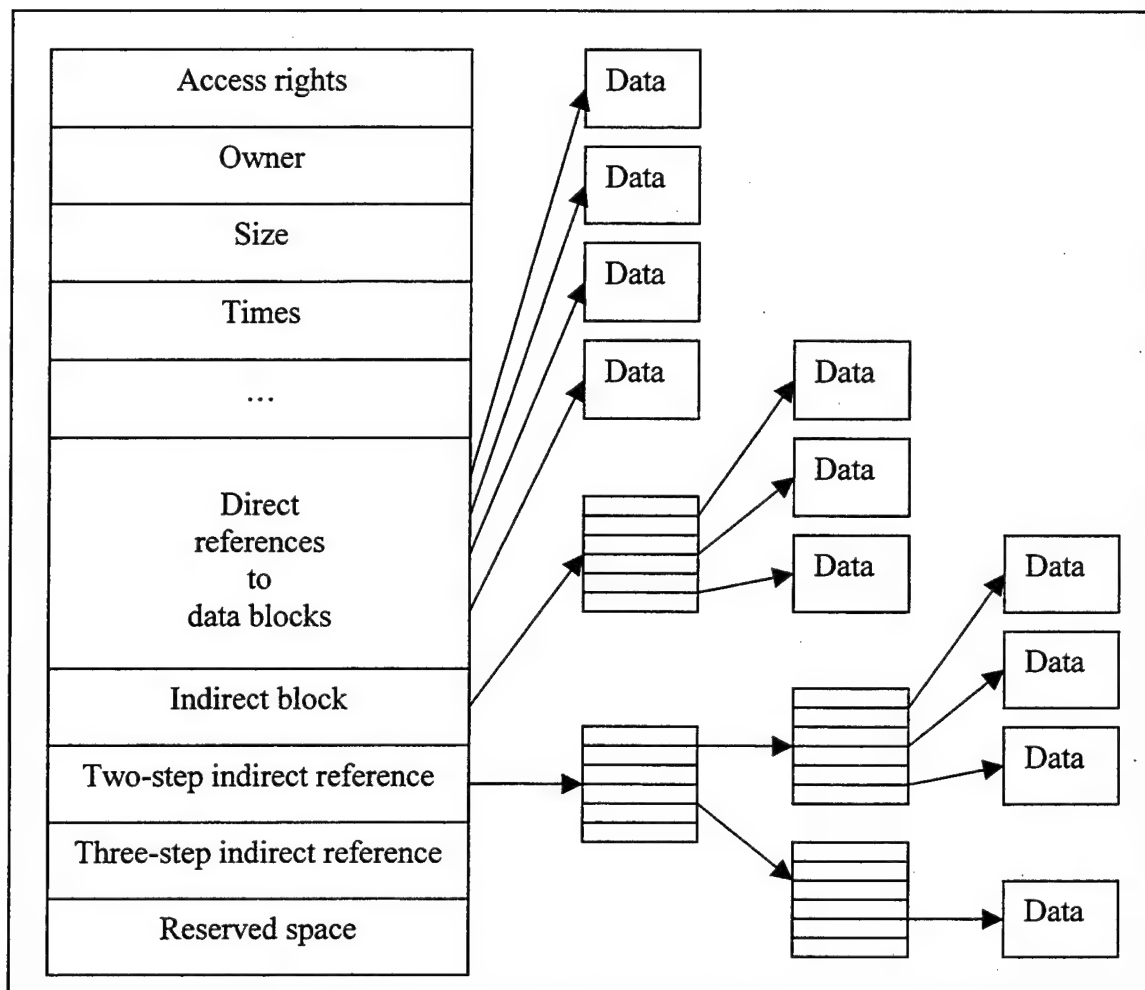


Figure 11. Structure of a Linux inode, After Ref. 12, p. 150

The non-MAC inode structure currently requires 128 bytes when compiled on an Intel CPU. Inodes are stored in disk blocks that are typically 1,024 bytes in size, allowing 8 inodes per block. The current structure of the inode has 8 bytes of reserved space that will be used for our purposes. [Ref 12, p.182] The initial implementation of the MAC policies will use these reserved bytes to store the Policy Label. Future implementations may expand the size of the inode in order to support a potentially large number of enforced policies.

3. File Statistics

Many programs need to obtain information about file system objects. A prime example is the command shell, when it needs to list the contents of the current directory. To get the necessary information, the shell makes one call to the operating system for each object in the current directory. The information returned for each object needs to include the Policy Label. This requires a change to the following data structure:

- struct stat

in addition to the following system calls:

- stat
- fstat
- lstat.

4. Subject Changes

Now that labels are associated with file system objects, it is necessary to design the other half of the policy-enforcement mechanism: associating Policy Labels with the subjects that access the objects. If a subject passes the usual Linux DAC check, it must then pass a MAC check such that the Policy Label of the object is compared with the Policy Label of the subject.

Every subject in Linux has a data structure associated with it called `task_struct`. It provides the information needed by the subject to run properly and information needed by the kernel to make DAC decisions. For example, it contains the User ID and Group ID of the subject, which are compared with the User ID and Group ID of the file system objects the process tries to access. This structure is the obvious location for associating Policy Labels with subjects.

It is necessary to give each subject two Policy Labels in order to support trusted subjects. The two Policy Labels assigned to each subject are called the Read Label and the Write Label, which have the following relationship:

$$\text{Write Label} \leq \text{Read Label}$$

The Read Label is the highest level that the subject can read, whereas the Write Label is the lowest level that the subject can write. For single-level (untrusted subjects) these two labels are equal.

5. Creating Objects

When a subject is running at a single level, objects created by the subject are assigned the same Policy Label as the subject. When a subject is a multilevel subject, the subject needs to communicate with the kernel to inform it of the Policy Label to assign to each new object. This can only be accomplished by modifying the `creat()` system call to accept the additional Policy Label parameter. This presents the following three requirements for creating new objects:

$$\text{Subject Read Class} \geq \text{Requested Object Class}$$

$$\text{Subject Write Class} \leq \text{Requested Object Class}$$

$$\text{Directory Class} \leq \text{Requested Object Class}$$

The first two restrictions simply mean that the multilevel subject must be able to both read and write at the requested access class of the new object. The latter restriction is given so the file hierarchy descends from the root to its leaves in non-decreasing levels. This restriction is necessary to avoid a “dangling object” that cannot be accessed by subjects at its level. This restriction is part of the original Bell and LaPadula secrecy policy, and is referred to as the “compatibility” property. [Ref. 8, p. 29]

Files are only classified at a single level. Directories, as objects that contain names and locations of other objects, are also single-level objects, but the objects that they point to may be at a different level than the level of the directory. However, there are constraints placed on the creation of objects whose class is different from that of its directory:

Subject Read Class \geq Requested Object's Directory

Subject Write Class \leq Requested Object's Directory

These last two restrictions mean that the multilevel subject must be able to both read and write to the new object's directory. Therefore, the only way to create an object in Policy Enhanced Linux with a different access class than that of the directory it is being created in, is by using a trusted subject, e.g., a multilevel process whose Read and Write class spans a range that includes the level of the directory and the level of the object being created. When an object is created with an access class that is higher than the directory it is being created in, it is referred to as an upgraded object.

Other designs have shown that it is possible for low-level subjects to create upgraded objects [Ref 13], but it would require a major redesign of how object statistics are stored and managed.

The following system calls have been identified as being affected by the changes described above:

- open
- creat
- mkdir
- symlink

6. Deleting Objects

Deleting an object requires the same privileges as Creating an object because it requires the modification of a directory. With respect to deleting objects, the following system calls are affected:

- unlink
- rmdir

7. Deflection Directories

Existing Linux applications expect to be able to read and write to a temporary directory, located at `"/tmp"` in the file system hierarchy. Because applications will be running at a number of levels simultaneously, and because these applications are mostly running as single-level subjects, this creates a problem, given the constraints described in the previous subsections. Fortunately, this is not a new problem.

When other multilevel Unix designs were faced with this problem of either modifying every existing application to write temporary files somewhere else, depending on the level they were running at, or coming up with an alternative, they all came up with an alternative. The best solution is something known as a Deflection Directory, first proposed by Kramer [Ref. 14, p. 28], though it is also known by other names [Ref. 15, p. 82][Ref. 16, p. 86][Ref. 17, p. 65].

A Deflection Directory is a directory that contains sub-directories for each level that is needed. Access to these directories is transparent to the applications. For example, if a subject at the SECRET level writes a temporary file to a deflection directory, say `"/tmp"`, the kernel will first create a SECRET sub-directory, transparent to the user, before creating the file in the SECRET sub-directory. From the user's point of view, the file was created in `"/tmp"` because the underlying system deflects every reference to `"/tmp"` to the `"/tmp/SECRET"` sub-directory for SECRET subjects. This allows existing applications to work without modification, no matter what level they are running. In order to prevent a covert channel, deflection directories can only be created and deleted by a System Administrator. See Appendix A for more information about this restriction with respect to covert channels.

In order to simplify the semantics of the deflection directory, only the System Administrator is exempt from the deflection. The administrator sees the true directory structure and can therefore list and access any file within a deflection hierarchy. This has the negative side effect of not allowing higher-level subjects to read the lower-level

objects in these directories. This is a trade-off between the complexity involved with determining when a subject wants to walk the deflected path, and when a subject wants to explicitly walk down a different path in the deflected hierarchy. See Chapter IV for additional information about alternative design decisions for deflection directories.

Over time, deflection directories can potentially grow quite large as new transparent directories are created. This leads to the design choice to delete all transparent directories under “/tmp” during system initialization.

The following system calls have been identified as being affected by the changes described above:

- Mkdir
- Chdir
- Fchdir
- Chroot
- open
- opendir
- stat
- lstat
- fstat

8. Updating Object Properties

Mandatory Access Control policies, such as the Bell and LaPadula policy, do not allow a high level subject to modify a file at a lower level; if this were allowed, a huge security hole would be created. A less obvious observation is that any change in an object's properties, such as the time of last access, creates a covert channel. Therefore, in addition to the usual MAC constraints, all object properties can only be changed by a subject at the same level of the object. This includes the following properties:

- Name
- Owner

- Group
- Size
- Time of last access.

The following system calls must be modified in order to selectively update object properties:

- rename
- truncate
- ftruncate
- chmod
- fchmod
- chown
- fchown
- utime
- utimes

Support for this design decision was only found in one publicly available document [Ref. 18, pp. 53-54], though the interface of both Trusted Solaris and the XTS-300 have a similar restriction. See Appendix A for more information about the related covert channel.

9. The Super User

Unix systems have a user known as the Super User, which is associated with any user who has a User ID value of 0, typically only given to a user with the name of "root." This user bypasses all security checks on the system. One becomes the super user in one of two ways: 1) logging in as the root user; 2) logging in as a regular user, executing the "su" (super user) command and entering the password for the root user.

This ability of the super user to bypass security checks will continue to be supported in Policy Enhanced Linux by including an exemption on the additionally

enforced policies. See Chapter IV for a presentation of additional work that can be done to improve security in this area.

Unix also supports something known as “setuid” and “setgid” programs, where executable files can be configured to run as the owner or group of the executing file, respectively, instead of running as the user who executes the file. This feature is typically used to allow a program to execute with root privileges, even when executed by a non-root user. Despite the fact that these features are recognized as a security weakness in Unix, Policy Enhanced Linux will not change how setuid and setgid programs work.

10. Setting Initial Policy Labels

Because the installation kernel will not (at least initially) be aware of labels, and therefore will not be setting the labels on the installed files, an installation of Policy Enhanced Linux will have uninitialized values in all object Policy Labels, as stored in their inodes. Several alternatives exist for solving this problem. The solution chosen for the initial implementation is to allow the root user to detect invalid labels and set them to a system low value. See Appendix A for other design choices.

11. Summary of Changes

The system calls and structures that need to be modified are summarized in Table

3.

System Calls		System Structures
access	mkdir	inode
chdir	open	task_struct
chmod	opendir	stat
chown	rename	
chroot	rmdir	
creat	stat	
fchdir	symlink	
fchmod	truncate	
fchown	utime	
fstat	utimes	
ftruncate	unlink	
lstat		

Table 3. Summary of Affected System Calls and Structures

B. APPLICATION MODIFICATIONS

1. File System Creation Program (mke2fs)

After a disk partition has been created and (optionally) formatted, the mke2fs utility is run on the partition to lay out the file system structures. If future implementations of Policy Enhanced Linux require a larger inode, then this utility must be re-compiled so it can create inodes of the proper size within a new partition.

2. Login Program (login)

The program called "login" presents the login interface to the user, checks a users password, and starts up the user environment if the password is correct. This program must be modified to also prompt the user for the desired session level. If no level is given, then the user's default session level is used. This session level must pass the following tests before the user environment is set up:

User's Minimum Session Level \leq Session Level

System low \leq Session Level \leq System High

Session Level \leq User's Clearance

The login program must also be modified to set the level of the user processes to the approved session level.

If a valid user of the system tries to log into the system, i.e., if a user has an entry in the "/etc/passwd" file, but does not have an entry in the Clearance Database, then the login will fail, with the exception of the root user.

3. Object Statistics (ls, stat)

There are two user-level programs that can be used to display statistics about file system objects: ls and stat. Both programs need to be modified to display the human-readable Policy Label when requested. It would be helpful for debugging purposes to have the stat command return either the human-readable label or an ASCII representation of the binary label. The ls command is sometimes implemented as an internal shell command, and would therefore require the modification of at least one shell, such as sh and csh.

The ls command must also be modified to indicate when a directory is a deflection directory when a full listing of a directory is made, i.e., when "ls -l" is used.

The 'd' that normally indicates that the entry is a directory must be replaced with a 'D' if the directory is a deflection directory.

4. Process Status (ps)

The ps command must be modified to have a new command-line option to allow the displaying of process Policy Labels.

5. Process Identification (id)

The id command must be modified to display the session level, in addition to the group ID and user ID that are currently displayed with this command.

6. Directory Creation (mkdir)

A new option (-M) must be added to the supported command-line options of the mkdir command. This new option will create the specified directory as a deflection directory. A deflection directory can only be created by a System Administrator, i.e., the root user.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. CONCLUSIONS

A. PROGRESS MADE

A minimal number of modules were implemented to show the feasibility of adding MAC policies to Linux. The following three (out of twelve) modules were either fully or partially implemented:

- Bell and LaPadula Policy (BLP_POL)
- Label Manager (LBL_MGR)
- Meta-Policy Manager (POL_MGR)

A small amount of Linux code was modified to support the addition of policy labels to subjects and objects, as well as the comparing of the labels to support the enforcement of the Bell and LaPadula secrecy policy. A demonstration of this new capability was produced, showing that a low-level subject could not read a high-level object, even when the DAC permissions allowed it. The remaining modules and Linux modifications can now be implemented with confidence that the system will work as designed.

B. PROBLEMS ENCOUNTERED

As with any programming project involving a large amount of unfamiliar code, such as porting a large application, a major hurdle was becoming familiar with the Linux code. The reference material was extremely helpful in identifying the areas that needed to be changed, and identifying some of the associated source files. [Ref. 12][Ref. 27][Ref. 28] Even with the reference material, however, there remained a lot of manual searching and reading of source files to determine where particular changes needed to be made, and to understand what the code was doing.

There were some initial problems of safely installing new instances of Policy Enhanced Linux onto a target partition. The new kernels had to be moved from the

development Linux partition to a separate target partition where the modified kernels were tested. In addition, initial installations of Policy Enhanced Linux would corrupt the target partition and require reinstallation. Eventually, procedures were developed to reduce turnaround time by making a backup image of a good target partition onto a third partition, so that it could be quickly restored. However, there was still concern that Policy Enhanced Linux would accidentally corrupt the development partition, or worse. Therefore, daily backups were made of all changes.

In terms of debugging, the `printk` internal kernel function was invaluable. It has the same syntax as the standard C `printf` function, but it does two things: 1) it prints the specified string to the console; and 2) it writes the same string into the system log, located at `"/var/log/messages."` At times there were so many debugging statements, that useful and efficient debugging could not have taken place without being able to digest the contents of the file from the development partition. There was one particular problem when a non-root user could not log in, and the login screen would clear before the debugging messages could be read; without the log, the problem would have taken longer to debug.

C. FUTURE RESEARCH

1. Additional User Roles

This research topic emphasized the inclusion of additional security policies into the Linux kernel. Another area of operating system security that needs to be addressed is that of additional user roles, to support the Principle of Least Privilege [Ref. 19]. This principle states that one should give a user (or process) only enough privilege to do his job, and no more [Ref. 4, p. 49] [Ref. 4, p. 286] [Ref 20, p. 378]. This is not a principle that is normally supported in a Unix environment.

Most Unix environments, including Linux, have only two privilege states: normal user and super user. It is an all or nothing approach to privilege. A more secure

environment would have a spectrum of privilege that could be granted to users, depending on what needed to be done. The following options could be researched in more depth:

- Remove the “su” command completely from the system and replace it with some other mechanism for obtaining dynamic privilege.
- Modify the “su” command to prompt the user for a particular user role, as well as a Read and Write class.

2. Auditing

Unix already supports a logging feature, but it is not considered an auditing system, with respect to security. A much more robust set of features must be designed and implemented to support the selective auditing of objects and subjects, with varying degrees of granularity. The Policy Label can be used to support this kind of auditing, but it may require the implementation of a new system call to allow that portion of the label to be modified for objects. User-level auditing should be straightforward.

A secure audit environment would also require the kind of separation of duties that was discussed in the previous subsection, so that a user can be assigned as the Audit Administrator. The Audit Administrator must have unique privileges to configure and monitor the auditing features and logs, while being able to prevent other system administrator roles from tampering with the audit trail.

An important part of the audit mechanism is a careful design of the audit record format. In addition, it is very important to consider how the audit log is used and managed, e.g., the user interface for reviewing the audit log in an effective and useful manner.

3. Setuid and Setgid Programs

Policy Enhanced Linux does not try to tackle the complicated issues surrounding Unix setuid and setgid programs. Because such programs are considered potential security problems, this is an area that can benefit greatly from innovative ideas. Future researchers should reference the efforts that have already been made in this area. [Ref. 21][Ref. 22][Ref. 23][Ref. 24]

4. Deflection Directories

The current design for deflection directories is very restrictive because it does not allow non-root users to access deflected directories at lower levels. To some, this may be considered a feature by only allowing subjects of a particular class to access deflected objects. However, in terms of user friendliness, a better approach seems appropriate. In addition, because the root user is not deflected, its temporary files are stored directly in “/tmp”, which may represent a security concern.

One approach could be to have a new option for the “ls” command that notifies Linux to try to not use deflection when finding the directory for the listing. However, in order to read the objects, changes must be made to somehow communicate to the open() system call an indicator that deflection should not be used. This could be done by either adding a new parameter to the open() call, or by having a special token that the user must specify at the beginning of the object’s path.

5. Administrative Interface

As described in Appendix B, the new databases for configuring the new policy interface are kept in text files. Changing a configuration requires a text editor and some working knowledge of what needs to be configured, or at least some good documentation. Such an interface is probably the preferred approach for experts but is fraught with disaster for any novice who may have to administer the system. Some work

needs to be done in the area of Human-Computer Interaction to produce a better interface to allow even a novice to correctly configure the new parts of the system associated with the use of security labels.

With respect to the text files, a more secure design might consider having the configuration files in a “hidden” part of the file system where only the administrative interfaces could view and modify them. However, these visible human-readable files have great educational value during hands-on exercises, allowing the user to gain a greater understanding of the underlying mechanisms.

6. Privileges

Experience with other label-based systems indicates there will be some applications that will not work in the restricted environment of Policy Enhanced Linux. One way to allow such applications to work, without running them as the root user, is to support a concept known as privileges. A privilege is an attribute that allows some aspect of the overall security policy to be bypassed. For example, an executable program could be configured to be exempt from the Mandatory Access Control policies. The more granular the privilege set is, the closer the design comes to supporting the Principle of Least Privilege. However, should such privileges be assigned to subjects, objects, or both?

7. Trusted Path

There currently is no guarantee in Linux that the login prompt presented to the user is coming from the real login program and not from a user program masquerading as the login program. It is trivial to write a program that looks like the login program and trick an unsuspecting user into entering a user name and password, which is then e-mailed to the malicious user. The mechanism for thwarting this attack is known as the Trusted Path, which is invoked by a special key sequence that cannot be intercepted by programs running in the user space. When the system sees this sequence, all I/O to the

monitor and keyboard are suspended, and the real login program is started. Windows NT uses the Control-Alt-Delete key sequence to start its Trusted Path. [Ref. 3, p. 171]

8. Policy Label Initialization

Some work needs to be done to provide a better way to initialize policy labels after Linux has been installed. The initial implementation lets the root user detect when a label is invalid and set it to system low. This is not a safe approach because a truly corrupted label will not be detected, and its label will be set to something that is visible to everyone on the system. The following list provides additional alternatives:

- Develop an off-line tool that can be used to set all the Policy Labels before the first boot of Policy Enhanced Linux. This appears to be the most insecure option because there will be a utility that can be used to modify label settings at will.
- Modify Policy Enhanced Linux to set all the Policy Labels to a system low label during its initial boot. Some thought must be given to how Linux will determine it is the initial boot in a way that it cannot be tricked into thinking it is the initial boot, when in fact it is not.
- Have a reserved user other than root, but with the same privileges, who can detect invalid labels and set them to system low. The difference in this approach versus the initial implementation is that the intended use of this user is only for installation purposes. The account can be deleted after initialization. This solves the problems associated with the root user.

9. Move or Port to the Newest Linux Kernel

Shortly after this thesis work was started, a new version of Linux was released. The completed code needs to be ported to the latest kernel for future development. The greatest concern, in terms of portability, is whether the newer version has added more

functionality to the EXT2 file system such that the hitherto reserved inode space has been depleted.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. DESIGN DECISIONS RELATED TO COVERT CHANNELS

A covert channel is a method of transferring data that was “not intended for information transfer at all.” [Ref. 26, p. 615] They are the bane of Mandatory Access Control (MAC) systems because they are the means for bypassing a MAC policy. [Ref 3, pp. 83-84] They exist because all subjects on a system share resources, such as a processor or secondary storage. By cleverly taking advantage of such resource sharing, a user can cause information to be downgraded, violating the enforced policy, and probably going unnoticed while doing it. The sections in this Appendix document design choices that were made in order to prevent or minimize covert channels. However, it must be pointed out once again that Policy Enhanced Linux is not a high-assurance system. In addition, although the modifications to Linux have been made with some care, there is no guarantee of robustness. The following two sections describe design decisions that were made to remove potential covert channels.

A. DEFLECTION DIRECTORIES

Policy Enhanced Linux was designed to limit the creation and deletion of deflection directories to System Administrators, i.e., the root user. This decision was made to avoid a well-known covert channel with respect to upgraded directories [Ref. 25, pp. 11-14]. If a subject at a lower level can create and delete a deflection directory, and if the system does not allow a subject to delete a directory with files in it, then a subject at a higher-level can signal bits of information to the lower-level subject by carefully timing the creation and/or deletion of files in a deflection directory to correspond to binary ones and zeros, respectively. One of the tutorials for the *Introduction to Computer Security* course demonstrates this principle on a Trusted Solaris system that has this design flaw.

There is another design option available for limiting this channel but it is not a “safe” option: a subject can be allowed to create deflection directories then later delete them, but the system cannot be allowed to return an error if the directory has files in it (say, at a higher level). This means that the user has more flexibility, but it allows a user

to accidentally delete a directory that has higher-level files in it without knowing it. This is clearly an undesirable side-effect. In addition, it does not completely close the channel because a user can possibly observe how long it takes to delete a very full directory versus an empty directory.

B. OBJECT PROPERTIES

The restriction on the changing of object properties to only those subjects that are running at the same level as the object was first introduced when considering the covert channel related to the modification of an object's last time of access. Whenever a file system object is accessed, the inode is updated with the time it occurred. In a multilevel environment, this should only be allowed when the subject accessing the object is at the same level as the object being accessed. It is obvious that a higher-level subject can read lower-level objects, but the fact that it happened should not be maintained by the operating system. Otherwise, a high-level subject could cooperate with a lower-level subject to transfer bits of information by carefully timing the access and/or non-access of files in such a way to correspond to binary ones and zeros, respectively. This weakness extends to all object properties. In fact, if object names are allowed to be changed from a higher level, then this would go beyond a covert channel into the realm of storage channel. Such a weakness is more of an overt channel than a covert channel [Ref. 26].

There is concern that some applications may fail because of these restrictions. One way to keep these restrictions, yet allow such applications an environment to run successfully, would be to introduce the concept of privileges. Privileges are a way to exempt subjects from portions of the underlying security mechanisms. In this example, the "broken" application could be assigned a privilege that grants it the ability to modify the object properties even when they exist at lower levels. Such privileges should only be assigned by a System Administrator, and with ample warning of the security risk that such a setting would create.

APPENDIX B. DATABASE DESIGN

This appendix provides a detailed description of the databases described in Chapter II. It is expected that the databases exist as text files that can be modified directly with a text editor, with the exception of the Policy Label, which is always in binary format. Therefore, numeric fields are entered as text strings that are then translated by the module into the corresponding binary values. This is done to make the administration of the system easier. In addition, these databases can have un-interpreted comments if the first character in a line has the '#' character. Figure 12 provides an example of a database with comments and "actual" values:

```
# Fictitious Database (1st commented line)
# The first non-commented line is given below:
VALUE1    VALUE2
```

Figure 12. Example of Comments in a Database

A. POLICY LABEL

Each object has a Policy Label associated with it, while each subject has two Policy Labels. These labels contain the fields shown Table 4.

Field	Description
Version	The version of the label.
SecLevel	The secrecy level. The greater the level, the greater the secrecy.
SecCats	The secrecy categories. Each bit represents a category that is On or Off.
IntLevel	The integrity level. The greater the level, the greater the integrity.
IntCats	The integrity categories. Each bit represents a category that is On or Off.
Reserved	Unused and reserved space.

Table 4. Policy Label Format

The Policy Label, with the fields defined in the previous table, has the structure defined in the Figure 13, according to the initial definition provided by the Label Manager. Data for additional policies can be added in the reserved field.

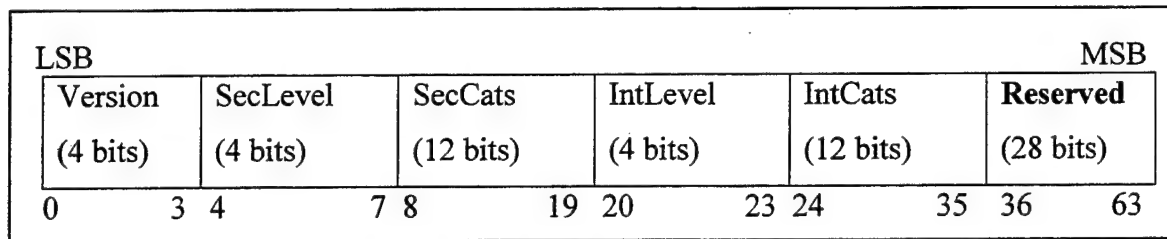


Figure 13. Policy Label Structure

B. LABEL DATABASE

There exists one Label Database for each enforced policy. With the exception of the version field, the label databases are policy-dependent. It is not possible to specify how a policy should map binary labels to human-readable labels without knowing something about the policy. The initial design of the Bell-LaPadula and Biba Label Databases is shown in Table 5.

Field	Description
Version	The version of the database. It is a non-negative integer on the first non-comment line of the database
Level	A Boolean value (1 or 0) indicating whether the binary/string combination that follows is for a level or a category: 1 = level, 0 = category.
Binary	A string representing a hexadecimal value that corresponds to a secrecy/integrity level or category.
String	The Human-Readable portion of the corresponding hexadecimal value.

Table 5. Policy-Dependent Portion of the BLP and Biba Label Databases

The Level, Binary and String fields are all on one line in the associated text file, separated by white space. These lines are repeated through the rest of the database. The database size is dynamic, depending on the number of levels and/or categories that are actually defined by an administrator.

An administrator can define levels and categories that are not actually being used, as long as they fall within the valid range of values defined by the policy. For example, if a system intends to use only four secrecy levels, an administrator may want to define the unused levels and categories with something descriptive, such as "UNUSED" or "INVALID".

C. USER CLEARANCE DATABASE

There exists one User-Clearance Database for each enforced policy. The information is stored in a human-readable format. This is necessary since the database will be modified via a text editor. After the version number, the remaining four fields in this database all occupy one line in the database per user. The four fields of the database are described in Table 6.

Field	Description
Version	The version of the database. It is a non-negative integer on the first non-commented line of the database.
UserID	The Unix User ID associated with the following clearances.
MinSession	In terms of a session level, the lowest level that the user can start a session at.
Clearance	In terms of a session level, the highest level that the user can start a session at.
DefaultSession	If a user does not specify a session level during the login sequence, he will be logged in at this session level.

Table 6. User Clearance Database

Each set of Human-Readable Labels in the database must satisfy the following relationship: $\text{MinSession} \leq \text{DefaultSession} \leq \text{Clearance}$.

D. RANGE DATABASE

There exists one Range Database for each enforced policy. Each database has three fields that are used to indicate the legal range of access that subjects can be given when executing on the system. Since this value can be changed during the lifetime of a system, there may be objects on the system that fall outside of this range. Table 7 describes these three fields.

Field	Description
Version	The version of the database. It is a non-negative integer on the first non-commented line of the database.
SysHigh	This is a human-readable label defining the highest level that any subject is allowed to write. This may in fact be lower than a user's MaxClearance. This value is on the second non-commented line of the database
SysLow	This is a human-readable label defining the lowest level that any subject is allowed to write. This may in fact be higher than a user's MinClearance. This value is on the third non-commented line in the database.

Table 7. Range Database

APPENDIX C. MODULE DESIGN

A. POLICY ENHANCED LINUX COMMON TYPES MODULE (PEL_TYP)

This module defines common types and constants used by other modules.

External Types and Constants:

```
typedef unsigned short    Bits16;
typedef unsigned long long Bits64;
typedef int               Boolean;

#define NO_ERROR          0
#define TRUE               1
#define FALSE             0
#define ALLOWED           TRUE
#define DISALLOWED        FALSE
#define VALID             TRUE
#define INVALID           FALSE
#define INCLUDED          TRUE
#define EXCLUDED          FALSE
```

B. BELL AND LAPADULA POLICY MODULE (BLP_POL)

This module defines the externally visible structure of the Bell and La-Padula secrecy label. The module interface is used to compare two secrecy labels to determine if one label dominates the other, to make queries about a given label, and to make changes to a label. This module does not depend on any other module.

External Entry Points:

- BlpPolInitLabel
- BlpPolSetLevel

- BlpPolGetLevel
- BlpPolAddCategory
- BlpPolDelCategory
- BlpPolTestCategory
- BlpPolDominates
- BlpPolRead
- BlpPolWrite

External Types and Constants:

```
#define MAX_SEC_LEVELS      16 /* max # of secrecy levels */
#define MAX_SEC_CATS        12 /* max # of secrecy categories */

/* error codes (decimal) */
#define BLPPOL_ERRBASE      1000
#define BLPPOL_BADLEVEL     BLPPOL_ERRBASE
#define BLPPOL_BADCATEGORY  BLPPOL_ERRBASE+1

typedef Bits16 BlpLabelType;
typedef Bits16 BlpLevelType;
typedef Bits16 BlpCatType;
```

1. BlpPollnitLabel

This entry point is used to initialize a BLP secrecy label.

a. External Interface

```
void BlpPollnitLabel( BlpLabelType *secLabel );
```

b. Inputs

<none>

c. Outputs

- **secLabel**
The initialized BLP secrecy label.

d. Processing

Call `BlpPolSetLevel`, passing in the lowest secrecy level as in `put`, and the input `secLabel` as output. For all categories (from 0 to `MAX_SEC_CATS-1`) call `BlpPolDelCategory`.

2. BlpPolSetLevel

This entry point is used to set the level portion of the BLP secrecy label to a given value.

a. External Interface

```
int BlpPolSetLevel(  
    const BlpLevelType  secLevel,  
    BlpLabelType        *secLabel  
);
```

b. Inputs

- **secLevel**
The secrecy level to put into the input `secLabel`.

c. Outputs

- **secLabel**

The label modified by setting its secrecy level field.

- **<function result>**

The success or failure of the operation. A value of NO_ERROR indicates success, while any other value indicates an error.

d. Processing

If the following relationship is TRUE, then continue:

$0 \leq \text{secLevel} < \text{MAX_SEC_LEVELS}$. Otherwise, return the BLPPOL_BADLEVEL error code as the function result.

Set the portion of the output secLabel that holds the secrecy level to the value stored in the input secLevel.

Return the value of NO_ERROR as the function result.

3. BlpPolGetLevel

This entry point is used to return the value of the current BLP secrecy level of a given BLP secrecy label.

a. External Interface

```
int BlpPolGetLevel(  
    const BlpLabelType  secLabel,  
    BlpLevelType        *secLevel  
);
```

b. Inputs

- `secLabel`

The secrecy label containing the secrecy level to be copied and returned to the caller.

c. Outputs

- `secLevel`

The secrecy level that is extracted from the input `SecLabel`.

- `<function result>`

The success or failure of the operation. A value of `NO_ERROR` indicates success, while any other value indicates an error.

d. Processing

Copy the portion of the input `secLabel` storing the secrecy level, and store the value in `tmpLevel`. If the following is TRUE, then continue:

$0 \leq \text{tmpLevel} < \text{MAX_SEC_LEVELS}$. Otherwise, return the `BLPPOL_BADLEVEL` error code as the function result.

Copy `tmpLevel` to the output `secLevel`, and return the value `NO_ERROR` as the function result.

4. BlpPolAddCategory

This entry point is used to add a particular category to the set of categories stored in a given BLP secrecy label. A category is referenced by its numerical value.

a. External Interface

```
int BlpPolAddCategory(  
    const BlpCatType  category,  
    BlpLabelType      *secLabel  
);
```

b. Inputs

- category

The specific category to add to the current set of categories in a secrecy label.

c. Outputs

- secLabel

The secrecy label with the given category added to its set of stored categories.

- <function result>

The success or failure of the operation. A value of NO_ERROR indicates success, while any other value indicates an error.

d. Processing

If the following relationship is TRUE, then continue:

$0 \leq \text{category} < \text{MAX_SEC_CATS}$. Otherwise, return the BLPPOL_BADCATEGORY error code as the function result.

Set the input category bit in the output secLabel to 1, then return the value of NO_ERROR as the function result.

5. BlpPolDelCategory

This entry point is used to delete a particular category from the set of categories stored in a given BLP secrecy label. A category is referenced by its numerical value.

a. External Interface

```
int BlpPolDelCategory(  
    const BlpCatType    category,  
    BlpLabelType        *secLabel  
);
```

b. Inputs

- category

The specific category to delete from the current set of categories in a secrecy label.

c. Outputs

- secLabel

The secrecy label with the given category deleted from its set of stored categories.

- <function result>

The success or failure of the operation. A value of NO_ERROR indicates success, while any other value indicates an error.

d. Processing

If the following relationship is TRUE, then continue:

$0 \leq \text{category} < \text{MAX_SEC_CATS}$. Otherwise, return the BLPPOL_BADCATEGORY error code as the function result.

Set the input category bit in the output secLabel to 0, then return the value of NO_ERROR as the function result.

6. BlpPolTestCategory

This entry point is used to test whether a particular category is currently in the set of stored categories in the given BLP secrecy label.

a. External Interface

```
int BlpPolTestCategory(  
    const BlpLabelType  secLabel,  
    const BlpCatType    category,  
    Boolean              *status  
);
```

b. Inputs

- secLabel
The secrecy label to use when testing whether the category is currently present.
- category
The specific category to look for.

c. Outputs

- status
A boolean value indicating whether the category is present (INCLUDED) or not (EXCLUDED).
- <function result>
The success or failure of the operation. A value of NO_ERROR indicates success, while any other value indicates an error.

d. Processing

If the following relationship is TRUE, then continue:

$0 \leq \text{category} < \text{MAX_SEC_CATS}$. Otherwise, set the output status to EXCLUDED and return the BLPPOL_BADCATEGORY error code as the function result.

If the input category bit in the input secLabel is turned on (1), then set the output status to INCLUDED. Otherwise, set the output status to EXCLUDED. Return the value of NO_ERROR as the function result.

7. BlpPolDominates

This entry point compares the first BLP secrecy label with the second secrecy label and communicates whether the first label dominates the second label.

a. External Interface

```
int BlpPolDominates(  
    const BlpLabelType  highLabel,  
    const BlpLabelType  lowLabel,  
    Boolean              *dominates  
);
```

b. Inputs

- **highLabel**

The BLP secrecy label to be tested for dominance against the input lowLabel.

- **lowLabel**

The BLP secrecy label to be tested for dominance against the input highLabel.

c. Outputs

- **dominates**

A value of TRUE or FALSE, indicating whether the input highLabel dominates the input lowLabel.

- **<function result>**

The success or failure of the operation. A value of NO_ERROR indicates success, while any other value indicates an error.

d. Processing

Get the secrecy level stored in the input highLabel by calling BlpPolGetLevel. If the function's return value is not equal to NO_ERROR, then return the value to the caller as this function's result (error). Otherwise, assign the returned level to levelHigh.

Get the secrecy level stored in the input lowLabel by calling BlpPolGetLevel. If the function's return value is not equal to NO_ERROR, then return the value to the caller as this function's result (error). Otherwise, assign the returned level to levelLow.

If $\text{levelHigh} < \text{levelLow}$, then assign FALSE to the output dominates, and return a value of zero (success) as the function result.

For every category bit from zero to $(\text{MAX_SEC_CATS}-1)$, do the following:

- Get the state of the category bit in the input highLabel by calling BlpPolTestCategory. If the function's return value is not equal to NO_ERROR, then return the value to the caller as this function's result (error). Otherwise, if INCLUDED is returned, then assign the TRUE to CatHigh. If EXCLUDED is returned, then assign FALSE to CatHigh.
- Get the state of the category bit in the input lowLabel by repeating the above step, replacing highLabel with lowLabel, and TRUE or FALSE to catLow.
- If the value of catHigh is FALSE (category off), and the value of catLow is TRUE (category on), then there is no domination. Set the output dominates to FALSE and return the value of NO_ERROR as the function result. Otherwise, continue.

If processing gets this far, then the input highLabel does dominate the input lowLabel. Set the output dominates to TRUE, and return the value of NO_ERROR as the function result.

8. BlpPolRead

This entry point determines whether read access is allowed by the secrecy policy, given the input subject and object secrecy labels.

a. External Interface

```
int BlpPolRead(  
    const BlpLabelType  subjectLabel,  
    const BlpLabelType  objectLabel,  
    Boolean              *access  
);
```

b. Inputs

- **subjectLabel**
The BLP secrecy label for the subject that wants to perform a read operation.
- **objectLabel**
The BLP secrecy label for the object to be read.

c. Outputs

- **access**
A value of ALLOWED or DISALLOWED, indicating whether the read operation requested by the associated subject is allowed for the associated object, or whether it should be disallowed.
- **<function result>**
The success or failure of the operation. A value of NO_ERROR indicates a success, while any other value indicates an error.

d. Processing

Determine if the input subjectLabel dominates the input objectLabel by calling BlpPolDominates, passing the input subjectLabel as the “highLabel”, and the

input objectLabel as the “lowLabel”. If the function’s return value is not equal to NO_ERROR, then return it as this function’s return value (error).

Otherwise, if the input subjectLabel does dominate the input objectLabel, then set the output access to ALLOWED. Otherwise, set the output access to DISALLOWED.

Return the value of NO_ERROR as the function result.

9. BlpPolWrite

This entry point determines whether a subject may have write access, with respect to the secrecy policy, given the input subject and object secrecy labels. A write operation is only allowed if both labels are valid and equal.

a. External Interface

```
int BlpPolWrite(  
    const BlpLabelType  subjectLabel,  
    const BlpLabelType  objectLabel,  
    Boolean              *access  
);
```

b. Inputs

- subjectLabel

The BLP secrecy label for the subject that wants to perform a write operation.

- objectLabel

The BLP secrecy label for the object to be modified.

c. Outputs

- **access**

A value of ALLOWED or DISALLOWED, indicating whether the write operation requested by the associated subject is allowed for the associated object, or disallowed.

- **<function result>**

The success or failure of the operation. A value of NO_ERROR indicates a success, while any other value indicates an error.

d. Processing

NOTE: this function does not just perform a bit-wise AND operation on the two labels to determine whether they are equal because the labels could be invalid. By using the BlpPolDominates, the labels are properly verified. If the two labels dominate each other, then they are equal.

Call BlpPolDominates, passing the input subjectLabel as the “highLabel”, and the input objectLabel as the “lowLabel”. If the function’s return value is not equal to NO_ERROR, then return it as this function’s return value (error). If subjectLabel does not dominate objectLabel, set the output access to DISALLOWED, and return NO_ERROR as the function result.

Call BlpPolDominates, passing the input objectLabel as the “highLabel”, and the input subjectLabel as the “lowLabel”. If the function’s return value is not equal to NO_ERROR, then return it as this function’s return value (error). If objectLabel does not dominate subjectLabel, set the output access to DISALLOWED, and return NO_ERROR as the function result.

If the labels dominate each other, then they are equal: set the output access to ALLOWED, and return NO_ERROR as the function result.

C. BIBA POLICY MODULE (BIB_POL)

This module defines the externally visible structure of the Biba integrity label. The module interface is used to compare two integrity labels to determine if one label dominates the other, to make queries about a given label, and to make changes to a label. This module does not depend on any other module.

External Entry Points:

- BibPolInitLabel
- BibPolSetLevel
- BibPolGetLevel
- BibPolAddCategory
- BibPolDelCategory
- BibPolTestCategory
- BibPolDominates
- BibPolRead
- BibPolWrite

External Types and Constants:

```
#define MAX_INT_LEVELS      16 /* max # of integrity levels */
#define MAX_INT_CATS        12 /* max # of integrity categories */

/* error codes (decimal) */
#define BIBPOL_ERRBASE      1100
#define BIBPOL_BADLEVEL     BIBPOL_ERRBASE
#define BIBPOL_BADCATEGORY  BIBPOL_ERRBASE+1

typedef Bits16 BibLabelType;
typedef Bits16 BibLevelType;
typedef Bits16 BibCatType;
```

1. BibPollInitLabel

This entry point is used to initialize a Biba integrity label.

a. External Interface

```
void BibPollInitLabel( BibLabelType *intLabel );
```

b. Inputs

<none>

c. Outputs

▪ intLabel

The initialized Biba integrity label.

d. Processing

Set all the bits in the output intLabel to zero.

2. BibPolSetLevel

This entry point is used to set the level portion of the Biba integrity label to a given value.

a. External Interface

```
int BibPolSetLevel(  
    const BibLevelType  intLevel,  
    BibLabelType        *intLabel  
);
```

b. Inputs

- `intLevel`

The integrity level to put into the input `intLabel`.

c. Outputs

- `intLabel`

The label modified by setting its integrity level field.

- `<function result>`

The success or failure of the operation. A value of `NO_ERROR` indicates success, while any other value indicates an error.

d. Processing

If the following relationship is TRUE, then continue:

$0 \leq \text{intLevel} < \text{MAX_INT_LEVELS}$. Otherwise, return the `BIBPOL_BADLEVEL` error code as the function result.

Set the portion of the output `intLabel` that holds the integrity level to the value stored in the input `intLevel`.

Return the value of `NO_ERROR` as the function result.

3. BibPolGetLevel

This entry point is used to return the value of the current Biba integrity level of a given Biba integrity label.

a. External Interface

```
int BibPolGetLevel(  
    const BibLabelType  intLabel,  
    BibLevelType        *intLevel  
);
```

b. Inputs

- intLabel
The integrity label containing the integrity level to be copied and returned to the caller.

c. Outputs

- intLevel
The integrity level that is extracted from the input intLabel.
- <function result>
The success or failure of the operation. A value of NO_ERROR indicates a success, while any other value indicates an error.

d. Processing

Copy the portion of the input intLabel that stores the integrity level, and store the value in tmpLevel. If the following is TRUE, then continue: $0 \leq \text{tmpLevel} < \text{MAX_INT_LEVELS}$. Otherwise, return the BIBPOL_BADLEVEL error code as the function result.

Copy tmpLevel to the output intLevel, and return NO_ERROR as the function result.

4. BibPolAddCategory

This entry point is used to add a particular category to the set of categories stored in the given Biba integrity label. A category is referenced by its numerical value.

a. External Interface

```
int BibPolAddCategory(  
    const CategoryType  category,  
    BibLabelType        *intLabel  
);
```

b. Inputs

- category

The specific category bit to add to the current set of categories in an integrity label.

c. Outputs

- intLabel

The integrity label with the given category added to its set of stored categories.

- <function result>

The success or failure of the operation. A value of NO_ERROR indicates a success, while any other value indicates an error.

d. Processing

If the following relationship is TRUE, then continue:

$0 \leq \text{category} < \text{MAX_INT_CATS}$. Otherwise, return the BIBPOL_BADCATEGORY error code as the function result.

Set the input category bit in the output intLabel to 1, then return the value NO_ERROR as the function result.

5. BibPolDelCategory

This entry point is used to delete a particular category from the set of categories stored in a given Biba integrity label. A category is referenced by its numerical value.

a. External Interface

```
int BibPolDelCategory(  
    const BibCatType  category,  
    BibLabelType      *intLabel  
);
```

b. Inputs

- **category**

The specific category to delete from the current set of categories in an integrity label.

c. Outputs

- `intLabel`

The integrity label with the given category deleted from its set of stored categories.

- `<function result>`

The success or failure of the operation. A value of `NO_ERROR` indicates a success, while any other value indicates an error.

d. Processing

If the following relationship is TRUE, then continue:

$0 \leq \text{category} < \text{MAX_INT_CATS}$. Otherwise, return the `BIBPOL_BADCATEGORY` error code as the function result.

Set the input category bit in the output `intLabel` to 0, then return the value `NO_ERROR` as the function result.

6. BibPolTestCategory

This entry point is used to test whether a particular category is currently in the set of stored categories in the given Biba integrity label.

a. External Interface

```
int BibPolTestCategory(  
    const BibaLabelType intLabel,  
    const CategoryType  category,  
    boolean              *status  
);
```

b. Inputs

- **intLabel**
The integrity label to use when testing whether the category is currently present.
- **category**
The specific category to look for.

c. Outputs

- **status**
A value of INCLUDED or EXCLUDED indicating whether the category is present (INCLUDED) or not (EXCLUDED).
- **<function result>**
The success or failure of the operation. A value of NO_ERROR indicates a success, while any other value indicates an error.

d. Processing

If the following relationship is TRUE, then continue:

$0 \leq \text{category} < \text{MAX_INT_CATS}$. Otherwise, set the output status to EXCLUDED, and return the BIBPOL_BADCATEGORY error code as the function result.

If the input category bit in the input intLabel is turned on (1), then set the output status to INCLUDED. Otherwise, set the output status to EXCLUDED. Return the value of NO_ERROR as the function result.

7. BibPolDominates

This entry point compares the first Biba integrity label with the second integrity label, returning TRUE if the first label dominates the second label. Otherwise it returns FALSE.

a. External Interface

```
int BibPolDominates(  
    const BibLabelType highLabel,  
    const BibLabelType lowLabel,  
    Boolean              *dominates  
);
```

b. Inputs

- highLabel
The Biba integrity label to be tested for dominance against the input lowLabel.
- lowLabel
The Biba integrity label to be tested for dominance against the input highLabel.

c. Outputs

- dominates
A boolean value of TRUE or FALSE, indicating whether the input highLabel dominates the input lowLabel.
- <function result>
The success or failure of the operation. A value of NO_ERROR indicates a success, while any other value indicates an error.

d. Processing

Get the integrity level stored in the input highLabel by calling BibaPolGetLevel. If the function's return value is not equal to NO_ERROR, then return the value to the caller as this function's result (error). Otherwise, assign the returned level to levelHigh.

Get the integrity level stored in the input lowLabel by calling BibaPolGetLevel. If the function's return value is not equal to NO_ERROR, then return the value to the caller as this function's result (error). Otherwise, assign the returned level to levelLow.

If $\text{levelHigh} < \text{levelLow}$, then assign FALSE to the output dominates and return the value of NO_ERROR as the function result.

For every category bit from zero to (MAX_INT_CATS-1), do the following:

- Get the state of the category bit in the input highLabel by calling BibaPolTestCategory [BIB_POL]. If the function's return value is not equal to NO_ERROR, then return the value to the caller as this function's result (error). Otherwise, if the category is present, assign TRUE to catHigh. If the category is not present, assign FALSE to catHigh.
- Get the state of the category bit in the input lowLabel by repeating the above step, replacing highLabel with lowLabel, and catHigh with catLow.
- If the value of catHigh is FALSE (category off), and the value of catLow is TRUE (category on), then there is no domination. Set the output dominates to FALSE and return the value of NO_ERROR as the function result. Otherwise, continue.

If processing gets this far, then the input highLabel does dominate the input lowLabel. Set the output dominates to TRUE, and return the value of NO_ERROR as the function result.

8. BibPolRead

This entry point determines whether read access is allowed by the Biba policy, given the input subject and object integrity labels.

a. External Interface

```
int BibPolRead(  
    const BibLabelType  subjectLabel,  
    const BibLabelType  objectLabel,  
    Boolean              *access  
);
```

b. Inputs

- subjectLabel
The Biba integrity label for the subject requesting read access.
- objectLabel
The Biba integrity label for the object to be read.

c. Outputs

- access
A value of ALLOWED or DISALLOWED, indicating whether the read operation requested by the associated subject is allowed for the associated object, or not.

- <function result>

The success or failure of the operation. A value of NO_ERROR indicates a success, while any other value indicates an error.

d. Processing

Determine if the input objectLabel does dominate the input subjectLabel by calling BibPolDominates, passing the input objectLabel as the “highLabel”, and the input subjectLabel as the “lowLabel”. If the function’s return value is not equal to NO_ERROR, then return it as this function’s return value (error).

Otherwise, if the input objectLabel does dominate the input subjectLabel, then set the output access to ALLOWED. Otherwise, set the output allowed to DISALLOWED.

Return the value of NO_ERROR as the function result.

9. BibPolWrite

This entry point determines whether a subject may have write access, with respect to the Biba policy, given the input subject and object integrity labels. A write operation is only allowed if both labels are equal.

a. External Interface

```
int BibPolWrite(
    const BibLabelType  subjectLabel,
    const BibLabelType  objectLabel,
    Boolean              *access
);
```

b. Inputs

- subjectLabel
The Biba integrity label for the subject requesting write access.
- objectLabel
The Biba integrity label for the object to be modified.

c. Outputs

- access
A value of ALLOWED or DISALLOWED, indicating whether the write operation requested by the associated subject is allowed for the associated object, or not.
- <function result>
The success or failure of the operation. A value of NO_ERROR indicates a success, while any other value indicates an error.

d. Processing

NOTE: this function does not just perform a bit-wise AND operation on the two labels to determine whether they are equal because the labels could be invalid. By using the BibPolDominates, the labels are properly verified. If the two labels dominate each other, then they are equal.

Call BibPolDominates, passing the input subjectLabel as the “highLabel”, and the input objectLabel as the “lowLabel”. If the function’s return value is not equal to NO_ERROR then return it as this function’s return value (error). If subjectLabel does not dominate objectLabel, set the output access to DISALLOWED, and return NO_ERROR as the function result.

Call BibPolDominates, passing the input objectLabel as the “highLabel”, and the input subjectLabel as the “lowLabel”. If the function’s return value is not equal to NO_ERROR, then return it as this function’s return value (error). If objectLabel does not dominate subjectLabel, set the output access to DISALLOWED, and return NO_ERROR as the function result.

If the labels dominate each other, then they are equal: set the output access to ALLOWED, and return NO_ERROR as the function result.

D. BELL AND LAPADULA LABEL MODULE (BLP_LBL)

This module is responsible for mapping binary Bell and LaPadula secrecy labels into human-readable secrecy labels (and vice versa). This module is dependent upon the BLP_POL module.

External Entry points:

- BlpLblInit
- BlpLblBinToStr
- BlpLblStrToBin

External types and constants:

```
#define MAX_LEVEL_STR          10
#define MAX_CAT_STR            10

/* error codes (decimal) */
#define BLPLBL_ERRBASE         1200
#define BLPLBL_NOFILE          BLPLBL_ERRBASE
#define BLPLBL_BADVERSION      BLPLBL_ERRBASE+1
#define BLPLBL_NOVERSION       BLPLBL_ERRBASE+2
#define BLPLBL_NOLEVELBITS     BLPLBL_ERRBASE+3
#define BLPLBL_BADLEVEL        BLPLBL_ERRBASE+4
```

```

#define BLPLBL_BADCATEGORY      BLPLBL_ERRBASE+5
#define BLPLBL_DUPCATEGORY     BLPLBL_ERRBASE+6
#define BLPLBL_DUPLEVELE      BLPLBL_ERRBASE+7
#define BLPLBL_LONGSTR         BLPLBL_ERRBASE+8
#define BLPLBL_UNDEFLEVEL      BLPLBL_ERRBASE+9
#define BLPLBL_UNDEFCATEGORY   BLPLBL_ERRBASE+10

```

Internal Databases:

```

/* Range of valid configuration database versions */

```

```

#define MIN_VERSION             1
#define MAX_VERSION             1

```

```

/* This structure holds a mapping of a secrecy level */

```

```

typedef struct {
    int    valid,
    int    level,
    char   subLabel[MAX_LEVEL_STR],
} LevelRecordType;

```

```

/* This structure holds a mapping of a secrecy category */

```

```

typedef struct {
    int    valid;
    int    category;
    char   subLabel[MAX_CAT_STR];
} CatRecordType;

```

```

/* The following comprise the internal representation of the */

```

```

/* BLP Label Database. */

```

```

LevelRecordType  levelMap[MAX_SEC_LEVELS];
CatRecordType    catMap[MAX_SEC_CATS];

```

1. BlpLblInit

This entry point is used to initialize the module during system startup when the system is still in single-user, single-process mode. Initialization includes opening the associated Label database, reading the contents into memory in internally static databases (for quick reference), then closing the file.

a. External Interface

```
void BlpLblInit(void);
```

b. Inputs

<none>

c. Outputs

<none>

When no errors are encountered, the procedure returns to the caller. If an error is encountered, an error message is displayed to the system console, and the system is halted.

d. Processing

Open the BLP Label database file, located at “/security/blrLabel”. If the file does not exist, display the BLPLBL_NOFILE error code and halt the system.

Read a line of text from the opened file until a line is read that does not contain a ‘#’ as the first character in the line (which represents a comment). This line contains the version number. If the following condition is true, then continue:

(version >= MIN_VERSION) AND (version <= MAX_VERSION)

Otherwise, display the BLPLBL_BADVERSION error code and halt the system. If the end of file is reached before a non-comment line is encountered, then display the BLPLABEL_NOVERSION error code and halt the system.

NOTE: the initial design only permits a version number with the value of one. As modifications are made and this field has more than one valid value, the processing below will become dependent on the version number.

Initialize the levelMap and catMap internal databases by setting the valid field in each entry of the database to FALSE.

Read a line of text from the opened file until another non-comment line is read. This line contains the level, binary and string fields of the database. If the value of the level field is "1" (indicating the string is for a level), do the following:

- Make sure that: $0 \leq \text{binary} < \text{MAX_SEC_LEVELS} [\text{BLP_POL}]$; if this relationship is not true, then display the error BLPLBL_BADLEVEL code and halt the system.
- Verify that the length of the string field is less than MAX_LEVEL_STR. If it is not, then display the BLPLBL_LONGSTR error code and halt the system.
- Look at each index of the levelMap array until the valid field of the array index is set to FALSE. For each valid entry, make sure the level field of the array index is not the same as the binary value just read from the file. If a duplicate is found, then return the BLPLBL_DUPLEVEl error code and halt the system.
- When an invalid entry is found, set the valid field to TRUE. Copy the value in the binary field into the level field of the array index. Copy the string value into the subLabel field of the array index.

Otherwise, if the value of the level field is "0" (indicating the string is for a category), then do the following:

- Make sure that: $0 \leq \text{binary} < \text{MAX_SEC_CATS}$ [from BLP_POL]; if this relationship is not true, then display the error BLPLBL_BADCATEGORY code and halt the system.
- Verify that the length of the string field is less than MAX_CAT_STR. If it is not, then display the error BLPLBL_LONGSTR code and halt the system.
- Look at each index of the catMap array until the valid field of the array index is set to FALSE. For each valid entry, make sure the category field of the array index is not the same as the binary value just read from the file. If a duplicate is found, then return the BLPLBL_DUPCATEGORY error code and halt the system.
- When an invalid entry is found, set the valid field to TRUE. Copy the value in the binary field into the category field of the array index. Copy the string value into the subLabel field of the array index.

Keep reading lines into the database as indicated in the previous paragraph, until the end-of-file is reached.

Close the opened file and return to the caller.

2. BlpLblBinToStr

This entry point is used to map a binary BLP secrecy label to a human-readable secrecy label.

a. External Interface

```
int BlpLblBinToStr(
    const BlpLabelType  binaryLabel,
    char                stringLabel[]
);
```

b. Inputs

- `binaryLabel`
The binary version of a BLP secrecy label.

c. Outputs

- `stringLabel`
The human-readable version of a BLP secrecy label.
- `<function result>`
If no errors are encountered during processing, then a value of `NO_ERROR` is returned as a function result. Otherwise, an error code is returned.

d. Processing

Get the secrecy level by calling `BlpPolGetLevel [BLP_POL]`, passing the input `binaryLabel`. If the returned value is not equal to `NO_ERROR`, then return that value as this function's return value (error). Otherwise, assign the returned secrecy level to `secLevel`.

Look at each valid field of each index of the `levelMap` array, comparing `secLevel` to the level field of the array index. When a match is found, copy the associated string from the array index to the start of the output `stringLabel`. If no match is found, then return the `BLPLBL_UNDEFLEVEL` error code as the function result.

For every category bit in the input `binaryLabel`, (0 to `MAX_SEC_CATS-1`) do the following:

- Determine whether the category bit is turned on by calling `BlpPolTestCategory [BLP_POL]`, passing the input `binaryLabel`. If

the function result is not equal to NO_ERROR, then return this value as this function's return value.

- Otherwise, if the bit is on, append a space to the end of the output stringLabel.
- Look at each valid field of each index of the catMap array, comparing the current category bit with the value stored in the category field of the array index. When a match is found, append the associated string from array index to the output stringLabel. If no match is found, then return the BLPLBL_UNDEFCATEGORY error code.

Return NO_ERROR as the function result.

3. BlpLblStrToBin

This entry point is used to map a human-readable BLP secrecy label to a binary BLP secrecy label.

a. External Interface

```
int BlpLblStrToBin(  
    const char    stringLabel[];  
    BlpLabelType  *binaryLabel;  
);
```

b. Inputs

- stringLabel
A human-readable version of a BLP secrecy label.

c. Outputs

- **binaryLabel**

The binary version of the input stringLabel.

- **<function result>**

If no errors are encountered during processing, then a value of NO_ERROR is returned as a function result. Otherwise, an error code is returned.

d. Processing

Get the first token in the input stringLabel. Compare its string of characters against all the values in the levelMap array whose corresponding valid field is set to TRUE. If a match is found at a particular array index, then set that portion of the output binaryLabel that holds the secrecy level to the category field of the same array index. If no match is found, then return the BLPLBL_BADLEVEL error.

Initialize the set of categories in the output binaryLabel by calling BlpPolDelCategory [BLP_POL] for each category (0 to MAX_SEC_CATS).

For all other tokens in the input stringLabel, compare their string of characters against the valid values in the catMap database one at a time. If a match is found at a particular array index, then call BlpPolAddCategory, passing the category field of the array index and the output binaryLabel. If no match is found, then return the BLPLBL_UNDEFCATEGORY error.

If all tokens are processed and no errors have been encountered, return NO_ERROR as the function result.

E. BIBA LABEL MODULE (BIB_LBL)

This module is responsible for mapping binary Biba integrity labels into human-readable integrity labels (and vice versa). This module is dependent upon the BIB_POL module.

External Entry points:

- BibLblInit
- BibLblBinToStr
- BibLblStrToBin

External types and constants:

```
#define MAX_LEVEL_STR      10
#define MAX_CAT_STR        10

/* error codes (decimal) */
#define BIBLBL_ERRBASE      1300
#define BIBLBL_NOFILE       BIBLBL_ERRBASE
#define BIBLBL_BADVERSION   BIBLBL_ERRBASE+1
#define BIBLBL_NOVERSION    BIBLBL_ERRBASE+2
#define BIBLBL_NOLEVELBITS  BIBLBL_ERRBASE+3
#define BIBLBL_BADLEVEL     BIBLBL_ERRBASE+4
#define BIBLBL_BADCATEGORY  BIBLBL_ERRBASE+5
#define BIBLBL_DUPCATEGORY  BIBLBL_ERRBASE+6
#define BIBLBL_DUPLEVELE    BIBLBL_ERRBASE+7
#define BIBLBL_UNDEFLEVEL   BIBLBL_ERRBASE+8
#define BIBLBL_UNDEFCATEGORY BIBLBL_ERRBASE+9
```

Internal Databases:

```
/* Range of valid configuration database versions */
#define MIN_VERSION    1
#define MAX_VERSION    1

/* This structure holds a mapping of an integrity level */
typedef struct {
    int    valid;
    char   subLabel[MAX_LEVEL_STR];
} LevelRecordType;

/* This structure holds a mapping of an integrity category */
typedef struct {
    int    valid;
    char   subLabel[MAX_CAT_STR];
} CatRecordType;

/* The following comprise the internal representation of the */
/* Biba Label Database */
LevelRecordType    levelMap[MAX_INT_LEVELS];
CatRecordType      catMap[MAX_INT_CATS];
```

1. BibLblInit

This entry point is used to initialize the module during system startup when the system is still in single-user, single-process mode. Initialization includes opening the associated Label database, reading the contents into memory in internally static databases (for quick reference), then closing the file.

a. External Interface

```
void BibLblInit(void);
```

b. Inputs

<none>

c. Outputs

<none>

When no errors are encountered, the procedure returns to the caller. If an error is encountered, an error message is displayed to the system console, and the system is halted.

d. Processing

Open the Biba Label database file, located at “/security/bibaLabel”. If the file does not exist, display the BIBLBL_NOFILE error code and halt the system.

Read a line of text from the opened file until a line is read that does not contain a ‘#’ as the first character in the line (which represents a comment). This line contains the version number. If the following condition is true, then continue:

(version >= MIN_VERSION) AND (version <= MAX_VERSION)

Otherwise, display the BIBLBL_BADVERSION error code and halt the system. If the end of file is reached before a non-comment line is encountered, then display the BIBLBL_NOVERSION error code and halt the system.

NOTE: the initial design only permits a version number with the value of one. As modifications are made and this field has more than one valid value, the processing below will become dependent on the version number.

Read a line of text from the opened file until another non-comment line is read. This line contains the level, binary and string fields of the database. If the value of the level field is "1" (indicating the string is for a level), do the following:

- Make sure that: $0 \leq \text{binary} < \text{MAX_INT_LEVELS [BIB_POL]}$; if this relationship is not true, then display the error BIBLBL_BADLEVEL code and halt the system.
- Verify that the length of the string field is less than MAX_LEVEL_STR. If it is not, then display the BIBLBL_LONGSTR error code and halt the system.
- Look at each index of the levelMap array until the valid field of the array index is set to FALSE. For each valid entry, make sure the level field of the array index is not the same as the binary value just read from the file. If a duplicate is found, then return the BIBLBL_DUPLELEVEL error code and halt the system.
- When an invalid entry is found, set the valid field to TRUE. Copy the value in the binary field into the level field of the array index. Copy the string value into the subLabel field of the array index.

Otherwise, if the value of the level field is "0" (indicating the string is for a category), then do the following:

- Make sure that: $0 \leq \text{binary} < \text{MAX_INT_CATS [BIB_POL]}$; if this relationship is not true, then display the error BIBLBL_BADCATEGORY code and halt the system.
- Verify that the length of the string field is less than MAX_CAT_STR. If it is not, then display the error BIBLBL_LONGSTR code and halt the system.

- Look at each index of the catMap array until the valid field of the array index is set to FALSE. For each valid entry, make sure the category field of the array index is not the same as the binary value just read from the file. If a duplicate is found, then return the BIBLBL_DUPCATEGORY error code and halt the system..
- When an invalid entry is found, set the valid field to TRUE. Copy the value in the binary field into the category field of the array index. Copy the string value into the subLabel field of the array index.

Keep reading lines into the database as indicated in the previous paragraph, until the end-of-file is reached.

Close the open file and return to the caller.

2. BibLblBinToStr

This entry point is used to map a binary Biba integrity label to a human-readable integrity label.

a. External Interface

```
int BibLblBinToStr(
    const BibLabelType  binaryLabel;
    char                stringLabel[];
);
```

b. Inputs

- binaryLabel
The binary version of a Biba integrity label.

c. Outputs

- **stringLabel**

The human-readable version of a Biba integrity label.

- **<function result>**

If no errors are encountered during processing, then a value of

NO_ERROR is returned as a function result. Otherwise, an error code is returned.

d. Processing

Get the integrity level by calling BibPolGetLevel [BIB_POL], passing the input binaryLabel. If the function result is not equal to NO_ERROR, then return that value as this function's return value. Otherwise, assign the returned integrity level to intLevel.

Look at each valid field of each index of the levelMap array, comparing intLevel to the level field of the array index. When a match is found, copy the associated string from the array index to the start of the output stringLabel. If no match is found, then return the BIBLBL_UNDEFLEVEL error code as the function result.

For every category bit in the input binaryLabel, (0 to MAX_INT_CATS-1) do the following:

- Determine whether the category bit is turned on by calling BibPolTestCategory [BIB_POL], passing the input binaryLabel. If the function result is not equal to NO_ERROR, then return this value as this function's return value.
- Otherwise, if the bit is on, append a space to the end of the output stringLabel.
- Look at each valid field of each index of the catMap array, comparing the current category bit with the value stored in the category field of

the array index. When a match is found, append the associated string from the array index to the output stringLabel. If no match is found, then return the BIBLBL_UNDEFCATEGORY error code.

Return NO_ERROR as the function result.

3. BibLblStrToBin

This entry point is used to map a human-readable Biba integrity label to a binary integrity label.

a. External Interface

```
int BibLblStrToBin(  
    const char        stringLabel[];  
    BibLabelType      *binaryLabel;  
);
```

b. Inputs

- stringLabel
A human-readable version of a Biba integrity label.

c. Outputs

- binaryLabel
The binary version of a Biba integrity label.
- <function result>
If no errors are encountered during processing, then a value of NO_ERROR is returned as a function result. Otherwise, an error code is returned.

d. Processing

Get the first token in the input stringLabel. Compare its string of characters against all the valid values in the levelMap array whose corresponding valid field is set to TRUE. If a match is found at a particular array index, then set that portion of the output binaryLabel that holds the integrity level to the category field of the same array index. If no match is found, then return the BIBLBL_UNDEFLEVEL error code as the function result.

Initialize the set of categories in the output binaryLabel by calling BibPolDelCategory [BIB_POL] for each category (0 to MAX_INT_CATS).

For all other tokens in the input stringLabel, compare their string of characters against the valid values in the catMap database one at a time. If a match is found at a particular array index, then call BibPolAddCategory, passing the category field of the array index and the output binaryLabel. If no match is found, then return the BIBLBL_UNDEFCATEGORY error.

If all tokens are processed and no errors have been encountered, return NO_ERROR as the function result.

F. BELL AND LAPADULA RANGE MODULE (BLP_RNG)

This module is responsible for managing the BLP range database, and returning the currently configured system high and system low secrecy labels for the system. This module is dependent on the Bell and LaPadula Policy (BPL_POL) and the Bell and LaPadula Label (BLP_LBL) modules.

External Entry points:

- BlpRngInit
- BlpRngSysLow
- BlpRngSysHigh

External types and constants:

```
/* error codes (decimal) */  
  
#define BLPRNG_ERRBASE      1400  
  
#define BLPRNG_NOFILE       BLPRNG_ERRBASE  
  
#define BLPRNG_BADVERSION   BLPRNG_ERRBASE+1  
  
#define BLPRNG_NOVERSION    BLPRNG_ERRBASE+2  
  
#define BLPRNG_NOSYSHIGH    BLPRNG_ERRBASE+3  
  
#define BLPRNG_NOSYSLOW     BLPRNG_ERRBASE+4
```

Internal Databases:

```
/* Range of valid configuration database versions */  
  
#define MIN_VERSION      1  
  
#define MAX_VERSION      1  
  
  
/* The following comprise the Internal representation of the */  
/* BLP Range Database */  
  
BlpLabelType      sysLow;  
BlpLabelType      sysHigh;
```

1. BlpRngInit

This entry point is used to initialize the BLP_RNG module when the system is still in single-user, single-process mode. It does this by reading a configuration file that stores the configured system high and system low secrecy labels for the BLP policy. It converts these human-readable labels into their binary form by using the BLP_LBL

module, and stores the result internally for future reference. If an error occurs during initialization, an error code is displayed and the system is halted.

a. External Interface

```
void BlpRngInit(void);
```

b. Inputs

- <none>

c. Outputs

- <none>

d. Processing

Open the BLP Range database file, located at “/security/blpRange”. If the file does not exist, display the error code BLPRNG_NOFILE and halt the system.

Read a line of text from the opened file until a line is read that does not contain a ‘#’ as the first character in the line (which represents a comment). This line contains the version number. If the following condition is true, then continue:

(version >= MIN_VERSION) AND (version <= MAX_VERSION).

Otherwise, display the BLPRNG_BADVERSION error code and halt the system. If the end of file is reached before a non-comment line is encountered, then display the BLPRNG_NOVERSION error code and halt the system.

NOTE: the initial design only permits a version number with the value of one. As modifications are made and this field has more than one valid value, the processing below will become dependent on the version number.

Read a line of text from the opened file until another non-comment line is read. This line contains the sysHigh field of the database. Assign this string to sysHighStr. If the end of file is reached before such a line is encountered, then display the BLPRNG_NOSYSHIGH error code and halt the system.

Read a line of text from the opened file until another non-comment line is read. This line contains the sysLow field of the database. Assign this string to sysLowStr. If the end of file is reached before such a line is encountered, then display the BLPRNG_NOSYSLOW error code and halt the system.

Call BlpLblStrToBin [BLP_LBL], passing in sysHighStr, and assign the returned Policy Label to the sysHigh module database. If BlpLblStrToBin returns a function result other than NO_ERROR, then display the returned error code and halt the system.

Call BlpLblStrToBin, passing in sysLowStr, and assign the returned Policy Label to the sysLow module database. If BlpLblStrToBin returns a function result other than NO_ERROR, then display the returned error code and halt the system. Otherwise, return to the caller.

Close the open file and return to the caller.

2. BlpRngSysLow

This entry point is used to return the binary system low BLP secrecy label.

a. External Interface

```
void BlpRngSysLow( BlpLabelType *systemLow );
```

b. Inputs

- <none>

c. Outputs

- systemLow

This is the currently configured binary system-low BLP secrecy label.

d. Processing

Copy the secrecy label stored in the sysLow database to the output systemLow, then return to the caller.

3. BlpRngSysHigh

This entry point is used to return the binary system high BLP secrecy label.

a. External Interface

```
void BlpRngSysHigh( BlpLabelType *systemHigh );
```

b. Inputs

- <none>

c. Outputs

- systemHigh

This is the currently configured binary system high BLP secrecy label.

d. Processing

Copy the secrecy label stored in the sysHigh database to the output systemHigh, then return to the caller.

G. BIBA RANGE MODULE (BIB_RNG)

This module is responsible for managing the Biba range database, and returning the currently configured system high and system low integrity labels for the system. This module is dependent on the Biba Policy (BIBS_POL) and the Biba Label (BIBA_LBL) modules.

External Entry points:

- BibRngInit
- BibRngSysLow
- BibRngSysHigh

External types and constants:

```
/* error codes (decimal) */  
#define BIBRNG_ERRBASE      1500  
#define BIBRNG_NOFILE       BIBRNG_ERRBASE  
#define BIBRNG_BADVERSION   BIBRNG_ERRBASE+1  
#define BIBRNG_NOVERSION    BIBRNG_ERRBASE+2  
#define BIBRNG_NOSYSHIGH    BIBRNG_ERRBASE+3  
#define BIBRNG_NOSYSLOW     BIBRNG_ERRBASE+4
```

Internal Databases:

```
/* Range of valid configuration database versions */  
#define MIN_VERSION      1  
#define MAX_VERSION      1
```

```
/* The following comprise the internal representation of the */
```

```
/* Biba Range Database */
```

```
BibLabelType sysLow;
```

```
BibLabelType sysHigh;
```

1. BibRngInit

This entry point is used to initialize the BIBA_RNG module when the system is still in single-user, single-process mode. It does this by reading a configuration file that stores the configured system high and system low integrity labels for the Biba policy. It converts these human-readable labels into their binary form by using the BIBA_LBL module, and stores the result internally for future reference. If an error occurs during initialization, an error code is displayed and the system is halted.

a. External Interface

```
void BibRngInit(void)
```

b. Inputs

- <none>

c. Outputs

- <none>

d. Processing

Open the Biba Range database file, located at “/security/bibaRange”. If the file does not exist, display the BIBARNG_NOFILE error code and halt the system.

Read a line of text from the opened file until a line is read that does not contain a '#' as the first character in the line (which represents a comment). This line contains the version number. If the following condition is true, then continue:

(version >= MAX_VERSION) AND (version <= MAX_VERSION).

Otherwise, display the BIBARNG_BADVERSION error code and halt the system. If the end of file is reached before a non-comment line is encountered, then display the BIBRNG_NOVERSION error code and halt the system.

NOTE: the initial design only permits a version number with the value of one. As modifications are made and this field has more than one valid value, the processing below will become dependent on the version number.

Read a line of text from the opened file until another non-comment line is read. This line contains the sysHigh field of the database. Assign this string to sysHighStr. If the end of file is reached before such a line is encountered, then display the BIBRNG_NOSYSHIGH error code and halt the system.

Read a line of text from the opened file until another non-comment line is read. This line contains the sysLow field of the database. Assign this string to sysLowStr. If the end of file is reached before such a line is encountered, then display the BIBRNG_NOSYSLOW error code and halt the system.

Call BibLblStrToBin [BIBA_LBL], passing in sysHighStr, and assign the returned Policy Label to the sysHigh database. If BibLblStrToBin returns a function result other than NO_ERROR, then display the returned error code and halt the system.

Call BibLblStrToBin, passing in sysLowStr, and assign the returned Policy Label to the sysLow database. If BibLblStrToBin returns a function result other than NO_ERROR, then display the returned error code and halt the system. Otherwise, return to the caller.

Close the opened file and return to the caller.

2. BibRngSysLow

This entry point is used to return the binary system low Biba integrity label.

a. External Interface

```
void BibRngSysLow( BibLabelType *systemLow );
```

b. Inputs

- <none>

c. Outputs

- systemLow

This is the currently configured binary system low Biba integrity label.

d. Processing

Copy the integrity label stored in the systemLow database to the output sysLow, then return to the caller.

3. BibRngSysHigh

This entry point is used to return the binary system high Biba integrity label.

a. External Interface

```
void BibRngSysHigh( BibLabelType *systemHigh );
```

b. Inputs

- <none>

c. Outputs

- systemHigh
This is the currently configured binary system high Biba integrity label.

d. Processing

Copy the integrity label stored in the sysHigh database to the output systemHigh, then return to the caller.

H. BELL AND LAPADULA CLEARANCE MODULE (BLP_CLR)

This module is responsible for managing the BLP clearance database, and for returning user clearance information upon request. There is no initialization point in this module because the user database could potentially be quite large, and the large amount of memory used by the kernel to keep an internal copy of the database would be an unwise use of resources. In addition, it is expected that clearance information would not be needed very often, and the design should not require the rebooting of the system just to accept a new user. Therefore, the configuration database is opened and searched upon each invocation of the entry point. This module is dependent on the BLP Policy (BLP_POL) and the BLP Label (BLP_LBL) modules.

External Entry points:

- BlpClrGetClearance

External types and constants:

```
/* error codes (decimal) */  
#define BLPCLR_ERRBASE      1600  
#define BLPCLR_NOFILE       BLPCLR_ERRBASE+0  
#define BLPCLR_BADVERSION   BLPCLR_ERRBASE+1  
#define BLPCLR_NOVERSION    BLPCLR_ERRBASE+2  
#define BLPCLR_NOTFOUND     BLPCLR_ERRBASE+3  
#define BLPCLR_BADCONFIG    BLPCLR_ERRBASE+4
```

Internal Databases:

```
/* Range of valid configuration database versions */  
#define MIN_VERSION      1  
#define MAX_VERSION      1
```

1. BlpClrGetClearance

This entry point is used to return the BLP clearance information for a particular user. It does this by opening and searching the BLP clearance database upon each request.

a. External Interface

```
int BlpClrGetClearance(  
    __uid_t      uid;  
    BlpLabelType *minSession;  
    BlpLabelType *clearance;  
    BlpLabelType *defaultSession  
);
```

b. Inputs

- uid

This is the User ID for the user in question.

c. Outputs

- MinSession

The lowest session level that a user can negotiate, with respect to the BLP policy.

- Clearance

The highest session level that a user can negotiate, with respect to the BLP policy.

- defaultSession

The default session level assigned to a user, with respect to the BLP policy.

- <function result>

The success or failure of the operation. A value of NO_ERROR indicates a success, while any other value indicates an error.

d. Processing

Open the BLP Clearance database file, located at “/security/blrClearance”.

If the file does not exist, display the error code BLPCLR_NOFILE.

Read a line of text from the opened file until a line is read that does not contain a ‘#’ as the first character in the line (which represents a comment). This line contains the version number. If the following condition is true, then continue:

(version >= MIN_VERSION) AND (version <= MAX_VERSION).

Otherwise, display the error code BLPCLR_BADVERSION. If the end of file is reached

before a non-comment line is encountered, then display the BLPCLR_NOVERSION error code.

NOTE: the initial design only permits a version number with the value of one. As modifications are made and this field has more than one valid value, the processing below will become dependent on the version number.

Read a line of text from the opened file until another non-comment line is read. Convert the first token of the line to a number and compare it to the input uid. If they are not equal then read another line. Continue reading lines until the uid's match. If the file is read and no match is made, then return the BLPCLR_NOTFOUND error code.

If a line is found where the two uid values match, then assign the three remaining tokens to the internal minSessionStr, clearanceStr and defaultSessionStr, respectively.

Convert minSessionStr to its corresponding binary value by calling BlpLblStrToBin [BLP_LBL]. If BlpLblStrToBin returns a function result other than NO_ERROR, then return it as this function's error code. Otherwise, assign the returned secrecy label to minSessionBin.

Convert clearanceStr to its corresponding binary value by calling BlpLblStrToBin. If BlpLblStrToBin returns a function result other than NO_ERROR, then return it as this function's error code. Otherwise, assign the returned secrecy label to clearanceBin.

Convert defaultSessionStr to its corresponding binary value by calling BlpLblStrToBin. If BlpLblStrToBin returns a function result other than NO_ERROR, then return it as this function's error code. Otherwise, assign the returned secrecy label to defaultSessionBin.

Make sure that the default session level dominates the minimum session level by calling `BlpPolDominates [BLP_POL]`, passing in `defaultSessionBin` as the first input and `minSessionBin` as the second argument. If a function result other than `NO_ERROR` is returned, then return it as this function's error code. Otherwise, if `defaultSessionBin` does NOT dominate `minSessionBin`, return the `BLPCLR_BADCONFIG` error code.

Make sure the user's clearance dominates the default session level by calling `BlpPolDominates`, passing in `clearanceBin` as the first input and `defaultSessionBin` as the second argument. If a function result other than `NO_ERROR` is returned, then return it as this function's error code. Otherwise, if `clearanceBin` does NOT dominate `defaultSessionBin`, return the `BLPCLR_BADCONFIG` error code.

Copy `minSessionBin` to the output `minSession`, and copy `clearanceBin` to the output `clearance`, and copy `defaultSessionBin` to the output `defaultSession`.

Close the open file, then return the value of `NO_ERROR` as this function's return value.

I. BIBA CLEARANCE MODULE (BIB_CLR)

This module is responsible for managing the Biba clearance database, and for returning user clearance information upon request. There is no initialization point in this module because the user database could potentially be quite large, and the large amount of memory used by the kernel to keep an internal copy of the database would be an unwise use of resources. In addition, it is expected that clearance information would not be needed very often, and the design should not require the rebooting of the system just to accept a new user. Therefore, the configuration database is opened and searched upon each invocation of the entry point. This module is dependent on the Biba Policy (`BIBA_POL`) and the Biba Label (`BIBA_LBL`) modules.

External Entry points:

- BibaClrGetClearance

External types and constants:

```
/* error codes (decimal) */  
#define BIBCLR_ERRBASE      1700  
#define BIBCLR_NOFILE       BIBCLR_ERRBASE  
#define BIBCLR_BADVERSION   BIBCLR_ERRBASE+1  
#define BIBCLR_NOVERSION    BIBCLR_ERRBASE+2  
#define BIBCLR_NOTFOUND     BIBCLR_ERRBASE+3  
#define BIBCLR_BADCONFIG    BIBCLR_ERRBASE+4
```

Internal Databases:

```
/* Range of valid configuration database versions */  
#define MIN_VERSION      1  
#define MAX_VERSION      1
```

1. BibaClrGetClearance

This entry point is used to return the Biba clearance information for a particular user. It does this by opening and searching the Biba clearance database upon each request.

a. External Interface

```
int BibaClrGetClearance(  
    __uid_t      uid;  
    BibLabelType *minSession;  
    BibLabelType *clearance;  
    BibLabelType *defaultSession  
);
```

b. Inputs

- uid

This is the User ID for the user in question.

c. Outputs

- MinSession

The lowest session level that a user can negotiate, with respect to the Biba policy.

- Clearance

The highest session level that a user can negotiate, with respect to the Biba policy.

- defaultSession

The default session level assigned to a user, with respect to the Biba policy.

- <function result>

The success or failure of the operation. A value of NO_ERROR indicates a success, while any other value indicates an error.

d. Processing

Open the Biba Clearance database file, located at “/security/bibaClearance”. If the file does not exist, display the error code BIBCLR_NOFILE.

Read a line of text from the opened file until a line is read that does not contain a ‘#’ as the first character in the line (which represents a comment). This line contains the version number. If the following condition is true, then continue:

(version >= MIN_VERSION) AND (version <= MAX_VERSION).

Otherwise, display the error code BIBCLR_BADVERSION. If the end of file is reached

before a non-comment line is encountered, then display the BIBCLR_NOVERSION error code.

NOTE: the initial design only permits a version number with the value of one. As modifications are made and this field has more than one valid value, the processing below will become dependent on the version number.

Read a line of text from the opened file until another non-comment line is read. Convert the first token of the line to a number and compare it to the input uid. If the are not equal then read another line. Continue reading lines until the uid's match. If the file is read and no match is made, then return the BIBCLR_NOTFOUND error code.

If a line is found where the two uid values match, then assign the three remaining tokens to the internal minSessionStr, clearanceStr and defaultSessionStr, respectively.

Convert the minSessionStr to its corresponding binary value by calling BibLblStrToBin [BIB_LBL]. If BibLblStrToBin returns a function result other than NO_ERROR, then return it as this function's error code. Otherwise, assign the returned integrity label to minSessionBin.

Convert clearanceStr into its corresponding binary value by calling BibLblStrToBin. If BibLblStrToBin returns a function result other than NO_ERROR, then return it as this function's error code. Otherwise, assign the returned integrity label to clearanceBin.

Convert defaultSessionStr into its corresponding binary value by calling BibLblStrToBin. If BibLblStrToBin returns a function result other than NO_ERROR, then return it as this function's error code. Otherwise, assign the returned integrity label to defaultSessionBin.

Make sure the default session label dominates the minimum session label by calling BibPolDominates [BIB_POL], passing in defaultSessionBin as the first input and minSessionBin as the second argument. If a function result other than NO_ERROR is returned, then return it as this function's error code. Otherwise, if defaultSessionBin does NOT dominate minSessionBin, return the BIBCLR_BADCONFIG error code.

Make sure the user clearance dominates the default session label by calling BibPolDominates, passing in clearanceBin as the first input and defaultSessionBin as the second argument. If a function result other than NO_ERROR, then return it as this function's error code. Otherwise, if clearanceBin does NOT dominate defaultSessionBin, return the BIBCLR_BADCONFIG error code.

Copy minSessionBin to the output minSession, and copy clearanceBin to the output clearance, and copy defaultSessionBin to the output defaultSession.

Close the open file, then return the value of NO_ERROR as this function's return value.

J. LABEL MANAGER (LBL_MGR)

This module is responsible for mapping human-readable Policy Labels into their binary counterparts, and vice versa. It is also responsible for extracting and setting the sub-labels in a binary Policy Label. This module is dependent on the modules in both the Label Layer and Policy Layer.

External Entry points:

- LblMgrInitLabel
- LblMgrBinToStr
- LblMgrStrToBin
- LblMgrGetBlp
- LblMgrSetBlp

- LblMgrGetBiba
- LblMgrSetBiba

External types and constants:

```
typedef Bits64 PolicyLabelType;
```

```
#define MAX_HRL_STR    256
```

```
#define DELIMITER      ':'
```

```
/* error codes */
```

```
#define LBLMGR_ERRBASE    1800
```

```
#define LBLMGR_BADSTRING  LBLMGR_ERRBASE
```

```
#define LBLMGR_BADLABEL   LBLMGR_ERRBASE+1
```

Internal Databases:

```
/* binary policy label version information */
```

```
#define MIN_VERSION      1
```

```
#define MAX_VERSION      15
```

```
#define CUR_VERSION      1
```

```
#define NUM_VERSION_BITS  4
```

1. LblMgrInitLabel

This entry point is used to initialize a Policy Label in two ways: 1) properly set the version field to avoid LBLMGR_BADLABEL errors; 2) getting initialized secrecy and integrity labels and storing them in the Policy Label.

a. External Interface

```
void LblMgrInitLabel( PolicyLabelType *binaryLabel );
```

b. Inputs

- <none>

c. Outputs

- binaryLabel
An initialized binary Policy Label.

d. Processing

Set all the bits in the output binaryLabel to zero.

Set the version field of the output label to the value of CUR_VERSION.

Call BlpPolInitLabel [BLP_POL] to initialize a secrecy label, then call LblMgrSetBlp to copy it into the output binaryLabel.

Call BibPolInitLabel [BIB_POL] to initialize an integrity label, then call LblMgrSetBiba to copy it into the output binaryLabel.

2. LblMgrIsValid

This entry point is used to test whether a binary Policy Label is valid or not.

a. External Interface

Boolean LblMgrIsValid(const PolicyLabelType binaryLabel);

b. Inputs

- **binaryLabel**

The binary Policy Label to test for validity.

c. Outputs

- **<result>**

A value of VALID is returned if the input binaryLabel is valid.

Otherwise, a value of INVALID is returned.

d. Processing

Copy the value from the version field in the input binaryLabel and assign it to curVersion. Test the relationship:

(curVersion >= MIN_VERSION) AND (curVersion <= CUR_VERSION). If the relationship is TRUE, then return VALID as the function result. Otherwise, return INVALID.

3. LblMgrBinToStr

This entry point is used to convert a binary Policy Label to its human-readable form.

a. External Interface

```
int LblMgrBinToStr(  
    const PolicyLabelType    binaryLabel,  
    char                     stringLabel[]  
);
```

b. Inputs

- **binaryLabel**
The binary Policy Label to convert to a human-readable format.

c. Outputs

- **stringLabel**
The human-readable version of the input binaryLabel.
- **<function result>**
The success or failure of the operation. A value of `NO_ERROR` indicates a success, while any other value indicates an error.

d. Processing

Copy the value from the version field in the input binaryLabel and assign it to curVersion. If the following condition is true, then continue:

`(curVersion >= MIN_VERSION) AND (curVersion <= CUR_VERSION).`

Otherwise, return the `LBLMBR_BADLABEL` error code as this function's return value.

NOTE: the initial design only permits a version number with the value of one. As modifications are made and this field has more than one valid value, the processing below will become dependent on the version number.

Initialize the output stringLabel so it has a length of zero.

- **BLP:**
Call `LblMgrGetBlp` to copy the BLP secrecy label from the input binaryLabel. Assign the output to secrecyBin.

Call BlpLblBinToStr to convert the secrecy label into a readable format, passing secrecyBin as input, and assigning the output to secrecyStr. If an error is returned, return the error as this function's return value.

If the following relationship is TRUE, then return the LBLMGR_BADSTRING error code as this function's return value:

$$\text{length}(\text{stringLabel}) + \text{length}(\text{secrecyStr}) > \text{MAX_HRL_STR}$$

Copy the secrecyStr to the end of the output stringLabel, followed by a space, the DELIMETER, then another space.

▪ **Biba:**

Call LblMgrGetBiba to copy the Biba integrity label from the input binaryLabel. Assign the output to integrityBin.

Call BibLblBinToStr to convert the integrity label into a readable format, passing integrityBin as input, and assigning the output to integrityStr. If an error is returned, return the error as this function's return value.

Copy the integrityStr to the end of the output stringLabel, followed by a null terminator to mark the end of the string.

Return the value of NO_ERROR as the function result.

4. LblMgrStrToBin

This entry point is used to convert a human-readable Policy Label to a binary Policy Label.

a. External Interface

```
int LblMgrStrToBin(  
    const char      stringLabel[],  
    PolicyLabelType *binaryLabel  
);
```

b. Inputs

- stringLabel
The human-readable version of a Policy Label.

c. Outputs

- binaryLabel
The binary version of the input stringLabel.
- <function result>
The success or failure of the operation. A value of NO_ERROR indicates a success, while any other value is an error.

d. Processing

If the following relationship is TRUE, return the LBLMGR_BADSTRING as this functions return value:

Length(stringLabel) > MAX_HRL_STR

Call LblMgrInitLabel to initialize the output binaryLabel.

- **BLP:**

Parse the input stringLabel to extract the human-readable BLP secrecy label, using the defined DELIMITER to differentiate the two policy strings. Assign the parsed string to secrecyString.

Convert secrecyString to its corresponding binary value by calling BlpLblStrToBin [BLP_LBL], using secrecyString as the input, and assigning the output to secrecyBinary. If an error is returned, return the error code as this function's return value.

Copy the secrecy label to the proper spot in the output binaryLabel by calling LblMgrSetBlp, passing in secrecyBinary as the input, and assigning the output to the output binaryLabel.

- **Biba:**

Parse the input stringLabel to extract the human-readable Biba integrity label, using the defined DELIMITER to differentiate the two policy strings. Assign the parsed string to integrityString.

Convert integrityString to its corresponding binary value by calling BibLblStrToBin [BIB_LBL], using integrityString as the input, and assigning the output to integrityBinary. If an error is returned, return the error code as this function's return value.

Copy the integrity label to the proper spot in the output binaryLabel by calling LblMgrSetBiba, passing in integrityBinary as the input, and assigning the output to the output binaryLabel.

Return the value of NO_ERROR as the function result.

5. LblMgrGetBlp

This entry point is used to copy the BLP portion of a Policy Label and return it to the caller.

a. External Interface

```
int LblMgrGetBlp(  
    const PolicyLabelType    policyLabel,  
    BlpLabelType             *secrecyLabel  
);
```

b. Inputs

- policyLabel
A binary Policy Label.

c. Outputs

- secrecyLabel
The BLP secrecy portion of the input Policy Label.
- <function result>
The success or failure of the operation. A value of NO_ERROR indicates a success, while any other value indicates an error.

d. Processing

Copy the version field from the input policyLabel to lblVersion. If the following condition is true, then continue:

(lblVersion >= MIN_VERSION) AND (lblVersion <= CUR_VERSION).

Otherwise, return the LBLMGR_BADLABEL error code as the function's return value.

NOTE: the initial design only permits a version number with the value of one. As modifications are made and this field has more than one valid value, the processing below will become dependent on the version number.

Copy the proper bits from the input policyLabel to the output secrecyLabel. The BLP secrecy label is just after the version bits. The number of bits to extract is equal to the number of bits in a secrecy label.

Return the value of NO_ERROR as the function's return value.

6. LblMgrGetBiba

This entry point is used to copy the Biba integrity portion of a Policy Label and return it to the caller.

a. External Interface

```
int LblMgrGetBiba(  
    const PolicyLabelType    policyLabel,  
    BibLabelType             *integrityLabel  
);
```

b. Inputs

- policyLabel
A binary Policy Label.

c. Outputs

- integrityLabel
The Biba integrity portion of the input Policy Label.

- <function result>

The success or failure of the operation. A value of NO_ERROR indicates a success, while any other value indicates an error.

d. Processing

Copy the version field from the input policyLabel to lblVersion. If the following condition is true, then continue:

(lblVersion >= MIN_VERSION) AND (lblVersion <= CUR_VERSION).

Otherwise, return the LBLMGR_BADLABEL error code as the function's return value.

NOTE: the initial design only permits a version number with the value of one. As modifications are made and this field has more than one valid value, the processing below will become dependent on the version number.

Copy the proper bits from the input policyLabel to the output integrityLabel. The Biba integrity label is just after the BLP label. The number of bits to extract is equal to the number of bits in an integrity label.

Return the value of NO_ERROR as the function's return value.

7. LblMgrSetBlp

This entry point is used to Set the BLP secrecy portion of a Policy Label.

a. External Interface

```
void LblMgrSetBlp(
    const BlpLabelType  secrecyLabel,
    PolicyLabelType     *policyLabel
);
```

b. Inputs

- `secrecyLabel`

The secrecy label to copy into the Policy Label.

c. Outputs

- `policyLabel`

A Policy Label with the input secrecy label properly installed.

d. Processing

Copy the version field from the input `policyLabel` to `lblVersion`. If the following condition is true, then continue:

$(\text{lblVersion} \geq \text{MIN_VERSION}) \text{ AND } (\text{lblVersion} \leq \text{CUR_VERSION})$.

Otherwise, return the `LBLMGR_BADLABEL` error code as the function's return value.

Copy the input `secrecyLabel` to the proper location in the output `policyLabel`. The BLP secrecy label is just after the version bits. The number of bits to extract is equal to the number of bits in a secrecy label.

8. LblMgrSetBiba

This entry point is used to Set the Biba integrity portion of a Policy Label.

a. External Interface

```
void LblMgrSetBiba(  
    const BibLabelType  integrityLabel,  
    PolicyLabelType     *policyLabel  
);
```

b. Inputs

- integrityLabel
The integrity label to copy into the Policy Label.

c. Outputs

- policyLabel
The Policy Label to use when inserting the input integrityLabel.

d. Processing

Copy the version field from the input policyLabel to lblVersion. If the following condition is true, then continue:

$(\text{lblVersion} \geq \text{MIN_VERSION}) \text{ AND } (\text{lblVersion} \leq \text{CUR_VERSION})$.

Otherwise, return the LBLMGR_BADLABEL error code as the function's return value.

Copy the input integrityLabel to the proper location in the output policyLabel. The Biba integrity label is stored just after the BLP label. The number of bits to extract is equal to the number of bits in an integrity label.

K. CLEARANCE MANAGER (CLR_MGR)

This module is responsible for returning user restrictions, with respect to session levels, in the form of Policy Labels. These labels are constructed using individual pieces returned by the modules in the Clearance Layer, then combining them into a binary Policy Label using the Label Manager. This module is also dependent on the modules in the Policy Layer.

External Entry points:

- ClrMgrGetClearance

External types and constants:

```
/* error codes (decimal)*/  
#define CLRMGR_NOUSER      1900
```

1. ClrMgrGetClearance

This entry point is used to return clearance information about a particular user.

a. External Interface

```
int ClrMgrGetClearance(  
    const __uid_t      uid,  
    PolicyLabelType     *minSession,  
    PolicyLabelType     *clearance,  
    PolicyLabelType     *defaultSession  
);
```

b. Inputs

- uid
The ID of the user in question.

c. Outputs

- minSession
The lowest session level the user can specify.
- clearance
The highest session level the user can specify.
- defaultSession
The default session level for the user.

- <function result>

The success or failure of the operation. A value of NO_ERROR indicates a success, while any other value indicates an error.

d. Processing

Initialize the output minSession, clearance and default Session by calling LblMgrInitLabel [LBL_MGR].

Call BlpClrGetClearance [BLP_CLR] to get the desired clearance information with respect to the BLP policy; pass the input uid as the function input, and assign the minimum session level, clearance, and default session level to blpSession, blpClearance and blpDefault, respectively. If an error is returned, then return the error as this function's error code.

Call BlpLblRange [BLP_LBL] to find out where the BLP secrecy label is stored in a Policy Label. Using this information, call LblMgrSetBlp [LBL_MGR] three times, passing the inputs and outputs as shown below:

- First call: pass blpSession and the output minSession.
- Second call: pass blpClearance and the output clearance.
- Third call: pass blpDefault and the output defaultSession.

Call BibClrGetClearance [BIB_CLR] to get the desired clearance information with respect to the Biba policy; pass the uid as the function input, and assign the minimum integrity level, clearance, and default session level to bibaSession, bibaClearance and bibaDefault, respectively. If an error is returned, then return the error as this function's error code.

Call BibLblRange [BIB_LBL] to find out where the Biba integrity label is stored in a Policy Label. Using this information, call LblMgrSetBiba [LBL_MGR] three times, passing the inputs and outputs as shown below:

- First call: pass bibaSession and the output minSession.
- Second call: pass bibaClearance and the output clearance.
- Third call: pass bibaDefault and the output defaultSession.

Return the value of NO_ERROR as the function result.

L. RANGE MANAGER (RNG_MGR)

This module is responsible for returning the currently configured system high and system low Policy Labels. These labels are constructed using individual pieces returned by the modules in the Range Layer, then combining them into a binary Policy Label by using the Label Manager. This module is also dependent on the modules in the Policy Layer.

External Entry points:

- RngMgrGetRange

1. RngMgrGetRange

This entry point is used to obtain the currently configured system high and system binary Policy Labels.

a. External Interface

```
void RngMgrGetRange(
    PolicyLabelType    *systemLow,
    PolicyLabelType    *systemHigh
);
```

b. Inputs

- <none>

c. Outputs

- systemLow
The system low Policy Label in binary format.
- systemHigh
The system high Policy Label in binary format.

d. Processing

Initialize the output systemLow and systemHigh by calling LblMgrInitLabel [LBL_MGR].

Call BlpRngSysLow [BLP_RNG] to get the system low information for the BLP policy, and assign the output to blpLow. Call BlpRngSysHigh [BLP_RNG] to get the system high information for the BLP policy, and assign the output to blpHigh.

Call BibRngSysLow [BIB_RNG] to get the system low information for the Biba policy, and assign the output to bibaLow. Call BibRngSysHigh [BIBA_RNG] to get the system high information for the Biba policy, and assign the output to bibaHigh.

Call BlpLblRange [BLP_LBL] to find out where the BLP secrecy label is stored in a Policy Label. Using this information, call LblMgrSetBlp [LBL_MGR] two times, passing the inputs and outputs as shown below:

- First call: pass blpLow and the output systemLow.
- Second call: pass blpHigh and the output systemHigh.

Call BibLblRange [BIB_LBL] to find out where the Biba integrity label is stored in a Policy Label. Using this information, call LblMgrSetBiba [LBL_MGR] two times, passing the inputs and outputs as shown below:

- First call: pass bibaLow and the output systemLow.
- Second call: pass bibaHigh and the output systemHigh.

M. META-POLICY MANAGER (POL_MGR)

This module is responsible for calling all of the modules in the Policy Layer to determine the dominance relationship between two binary Policy Labels, and to determine whether a given type of access is allowed.

External Entry points:

- PolMgrDominates
- PolMgrRead
- PolMgrWrite

1. PolMgrDominates

This entry point is used to test whether one Policy Label dominates another.

a. External Interface

```
int PolMgrDominates(  
    const PolicyLabelType    highLabel,  
    const PolicyLabelType    lowLabel,  
    Boolean                  *dominates  
);
```

b. Inputs

- **highLabel**
The Policy Label to use when testing for dominance.
- **lowLabel**
The Policy Label to use as the dominated label.

c. Outputs

- **dominates**
TRUE indicates that highLabel dominates lowLabel. FALSE indicates that highLabel does not dominate lowLabel. It cannot be assumed that lowLabel dominates highLabel if FALSE is returned.
- **<function result>**
The success or failure of the operation. A value of NO_ERROR indicates a success, while any other value indicates an error.

d. Processing

Call LblMgrGetBlp [LBL_MGR] to extract the BLP secrecy label portion of the input lowLabel, and assign the output to blpLow. If an error is returned, return that error as this function's return value.

Call LblMgrGetBlp to extract the BLP secrecy label portion of the input highLabel, and assign the output to blpHigh. If an error is returned, return that error as this function's return value.

Call LblMgrGetBiba [LBL_MGR] to extract the Biba integrity label portion of the input systemLow, and assign the output to bibaLow. If an error is returned, return the error as this function's return value.

Call LblMgrGetBiba to extract the Biba integrity label portion of the input systemHigh, and assign the output to bibaHigh. If an error is returned, return that error as this function's return value.

Call BlpPolDominates [BLP_POL], passing blpHigh and blpLow, and assign the output to blpLowDom. If an error is returned, return the error code as this function's return value.

Call BibPolDominates [BIB_POL], passing bibaHigh and bibaLow, and assign the output to bibaLowDom. If an error is returned, return the error code as this function's return value.

If both blpLowDom and bibaLowDom are set to TRUE, then the input highLabel does dominate the input lowLabel, so set the output dominates to TRUE. Otherwise, set it to FALSE.

Return the value of NO_ERROR as this function's return value.

2. PolMgrRead

This entry point is used to determine whether the associated subject should be allowed to read the associated object, given the subject and object Policy Labels. The operation is allowed if the input subjectLabel dominates the input objectLabel.

a. External Interface

```
int PolMgrRead(  
    const PolicyLabelType    subjectLabel,  
    const PolicyLabelType    objectLabel,  
    Boolean                  *access  
);
```

b. Inputs

- **subjectLabel**
The binary Policy Label representing the level of the subject that wants to read the object.
- **objectLabel**
The binary Policy Label representing the level of the object to be read.

c. Outputs

- **access**
A value of ALLOWED is returned if the subject associated with the input subjectLabel is allowed to read the object associated with the input objectLabel. Otherwise, a value of DISALLOWED is returned.
- **<function result>**
The success or failure of the operation. A value of NO_ERROR indicates a success, while any other value indicates an error.

d. Processing

Call PolMgrDominates, passing the input subjectLabel as the HIGH label, and the input objectLabel as the LOW label. If an error is returned, return the error code as this function's return value. If subjectLabel dominates objectLabel, then set the output access to ALLOWED. Otherwise, set the output access to DISALLOWED.

Return the value NO_ERROR as this function's return value.

3. PolMgrWrite

This entry point is used to determine whether the associated subject should be allowed to write the associated object, given the subject and object Policy Labels. The

only way that write permission will be allowed is if the subject and object labels are equal, viz., the two labels dominate each other.

a. External Interface

```
int PolMgrWrite(  
    const PolicyLabelType    subjectLabel,  
    const PolicyLabelType    objectLabel,  
    Boolean                  *access  
);
```

b. Inputs

- subjectLabel
The binary Policy Label representing the level of the subject that wants to write the object.
- objectLabel
The binary Policy Label representing the level of the object to be modified.

c. Outputs

- access
A value of ALLOWED is returned if the subject associated with the input subjectLabel is allowed to write the object associated with the input objectLabel. Otherwise, a value of DISALLOWED is returned.

d. Processing

NOTE: this function does not just perform a bit-wise AND operation on the two labels to determine whether they are equal because the labels could be invalid.

By using PolMgrDominates, the labels are properly verified. If the two labels dominate each other, then they are equal.

Call PolMgrDominates, passing the input subjectLabel as the HIGH label, and the input objectLabel as the LOW label. Assign the output to dominate1. If an error is returned, return the error code as this function's return value.

Call PolMgrDominates, passing the input objectLabel as the HIGH label, and the input subjectLabel as the LOW label. Assign the output to dominate2. If an error is returned, return the error code as this function's return value.

If dominate1 and dominate2 are both TRUE, then set the output access to ALLOWED. Otherwise, set the output access to DISALLOWED.

Return the value of NO_ERROR as this function's return value.

APPENDIX D. SOURCE CODE

This appendix contains the source code for the new modules that were implemented for the Policy Enhanced Linux. Each section is devoted to a module, with each subsection corresponding to individual source and header files.

A. POLICY ENHANCED LINUX COMMON TYPES (PEL_TYP)

1. Peltyp.h

```
// -----
// File: peltyp.h
// -----
// Description: This file contains types and constants that are common
// to
//     all Policy Enhanced Linux (PEL) modules.
//
// Created: 29-Jul-99 (P. Clark)
//
// Modifications:
// -----

#ifndef _PELTYP_H
#define _PELTYP_H

#define MAX_TRUSTED_UID 99
#define MAX_TRUSTED_PID 300

#define NO_ERROR 0
#define TRUE 1
#define FALSE 0
#define ALLOWED TRUE
#define DISALLOWED FALSE
#define VALID TRUE
#define INVALID FALSE

typedef unsigned short Bits16;
typedef unsigned long long Bits64;
typedef unsigned char Boolean;

#endif

// EOF for peltyp.h
```

B. BELL AND LAPADULA POLICY (BLP_POL)

1. blppol.h

```
// -----  
// File: bibpol.h  
// -----  
// Description: This file contains the external interface to the Biba  
// Policy Module (BIB_POL).  
//  
// Created: 04-Aug-99 (P. Clark)  
//  
// Modifications:  
// -----  
  
#ifndef _BIBPOL_H_  
#define _BIBPOL_H_  
  
#include <pel/peltyp.h>  
  
#define MAX_INT_LEVELS 16  
#define MAX_INT_CATS 12  
  
typedef Bits16 BibLabelType;  
typedef Bits16 BibLevelType;  
typedef Bits16 BibCatType;  
  
extern int BibPolSetLevel( const BibLevelType intLevel,  
                           BibLabelType *intLabel);  
  
extern int BibPolGetLevel( const BibLabelType intLabel,  
                           BibLevelType *intLevel);  
  
extern int BibPolAddCategory( const BibCatType category,  
                              BibLabelType *intLabel);  
  
extern int BibPolDelCategory( const BibCatType category,  
                              BibLabelType *intLabel);  
  
extern int BibPolTestCategory( const BibLabelType intLabel,  
                               const BibCatType category,  
                               Boolean *inTheSet);  
  
extern int BibPolDominates( const BibLabelType highLabel,  
                             const BibLabelType lowLabel,  
                             int *dominates);  
  
extern int BibPolRead( const BibLabelType subjectLabel,  
                       const BibLabelType objectLabel,  
                       int *allowed);
```

```

extern int BibPolWrite(      const BibLabelType  subjectLabel,
                             const BibLabelType  objectLabel,
                             int                  *allowed);

// Error codes returned by this module
//
#define BIBPOL_ERRBASE      1100
#define BIBPOL_BADLEVEL     BIBPOL_ERRBASE
#define BIBPOL_BADCATEGORY  BIBPOL_ERRBASE+1

#endif

// EOF for bibpol.h

```

2. blppol_getset.c

```

// -----
// File: blppol_getset.c
// -----
// Description: This is the implementation file for the following
// external entry points of the Bell and LaPadula Policy Module
// (BLP_POL):
// 1. BlpPolInitLabel
// 2. BlpPolSetLevel
// 3. BlpPolGetLevel
// 4. BlpPolAddCategory
// 5. BlpPolDelCategory
// 6. BlpPolTestCategory
//
// BLP binary label format:
//
// +-----+-----+
// |      categories      | level |
// +-----+-----+
// | 15          4 3    0  | <-- bits
//
// A label has a level field and a category field. There are 12
// possible categories. If a bit in the category field is on (1),
// then that category is in the set of categories stored by label.
// Otherwise, the bit is off (0) and the corresponding category
// is not in the set.
//
// Created: 29-Jul-99 (P. Clark)
//
// Modifications:
// -----
#include <pel/blppol.h>

```

```

#define NUM_LEVEL_BITS 4
#define ERASE_LEVEL 0xfff0
#define ERASE_CATS 0x000f
#define LOW_LEVEL 0

// -----
// Function:
//   BlpPolInitLabel
// Inputs:
//   <none>
// Outputs:
//   secLabel    The initialized BLP secrecy label.
//   <result>    The success or failure of the operation. A value of
//               zero indicates a success, while any other value
//               indicates an error.
// Description:
//   This entry point is used to initialize a BLP secrecy label to the
//   lowest level, with no categories (lowest secrecy).
// -----
int BlpPolInitLabel( BlpLabelType *secLabel )
{
    int          success    = NO_ERROR;
    BlpLabelType tmpLabel;
    BlpCatType   i;

    success = BlpPolSetLevel( LOW_LEVEL, &tmpLabel );
    if (success == NO_ERROR) {
        for (i=0; (i < MAX_SEC_CATS) && (success == NO_ERROR); ++i)
        {
            success = BlpPolDelCategory( i, &tmpLabel );
        }

        if (success == NO_ERROR) {
            *secLabel = tmpLabel;
        }
    }

    return(success);
} // BlpPolInitLabel

// -----
// Function:
//   BlpPolSetLevel
// Inputs:
//   secLevel    The secrecy level to put into the input secLabel.
// Outputs:
//   secLabel    The label to modify by setting its secrecy level.
//   <result>    The success or failure of the operation. A value of
//               NO_ERROR indicates a success, while any other value
//               indicates an error.
// Description:
//   This entry point is used to set the level portion of the BLP
//   secrecy level to a given value.
// -----

```

```

int BlpPolSetLevel( const BlpLevelType  secLevel,
                    BlpLabelType *secLabel )
{
    int success = NO_ERROR;

    if (secLevel < MAX_SEC_LEVELS) {
        // Get rid of the current level in the label
        *secLabel = (*secLabel) & ERASE_LEVEL;

        // Now put the new level in
        *secLabel = (*secLabel) | ((BlpLabelType) secLevel);
    }
    else {
        success = BLPPOL_BADLEVEL;
    }

    return(success);
} // BlpPolSetLevel

```

```

// -----
// Function:
//   BlpPolGetLevel
// Inputs:
//   secLabel      The secrecy label containing the secrecy level to be
//                 copied and returned to the caller.
// Outputs:
//   secLevel      The secrecy level that is extracted from the input
//                 secLabel.
//   <result>      The success or failure of the operation.  A value of
//                 NO_ERROR indicates a success, while any other value
//                 indicates an error.
// Description:
//   This entry point is used to return the current BLP secrecy level
//   of a given BLP secrecy label.
// -----

```

```

int BlpPolGetLevel( const BlpLabelType  secLabel,
                    BlpLevelType *secLevel )
{
    int          success = NO_ERROR;
    BlpLevelType tmpLevel;

    // Get rid of any categories that are turned on
    tmpLevel = ((BlpLevelType) secLabel) & ERASE_CATS;

    if (tmpLevel < MAX_SEC_LEVELS) {
        *secLevel = tmpLevel;
    }
    else {
        success = BLPPOL_BADLEVEL;
    }

    return(success);
} // BlpPolGetLevel

```

```

// -----
// Function:
//   BlpPolAddCategory
// Inputs:
//   category      The specific category to add to the current set of
//                  categories in a secrecy label.
// Outputs:
//   secLabel      The secrecy label with the given category added to
//                  its set of stored categories.
//   <result>      The success or failure of the operation.  A value of
//                  NO_ERROR indicates a success, while any other value
//                  indicates an error.
// Description:
//   This entry point is used to add a particular category to the set
//   of categories stored in a given BLP secrecy label.  A category is
//   referenced by its numerical value.
// -----
int BlpPolAddCategory( const BlpCatType   category,
                      BlpLabelType *secLabel )
{
    int          success = NO_ERROR;
    BlpLabelType tmpLabel;

    if (category < MAX_SEC_CATS) {
        tmpLabel = 0x0001;
        tmpLabel = tmpLabel << (NUM_LEVEL_BITS + category);
        *secLabel = (*secLabel) | tmpLabel;
    }
    else {
        success = BLPPOL_BADCATEGORY;
    }

    return(success);
} // BlpPolAddCategory

```

```

// -----
// Function:
//   BlpPolDelCategory
// Inputs:
//   category      The specific category to delete from the current set
//                  of categories in a secrecy label.
// Outputs:
//   secLabel      The secrecy label with the given category deleted
//                  from its set of stored categories.
//   <result>      The success or failure of the operation.  A value of
//                  NO_ERROR indicates a success, while any other value
//                  indicates an error.
// Description:
//   This entry point is used to delete a particular category to the
//   set of categories stored in a given BLP secrecy label.  A
//   category is referenced by its numerical value.
// -----
int BlpPolDelCategory( const BlpCatType   category,
                      BlpLabelType *secLabel )
{

```

```

    int          success = NO_ERROR;
    BlpLabelType tmpLabel;

    if (category < MAX_SEC_CATS) {
        tmpLabel = 0x0001;
        tmpLabel = ~(tmpLabel << (NUM_LEVEL_BITS + category));
        *secLabel = (*secLabel) & tmpLabel;
    }
    else {
        success = BLPPOL_BADCATEGORY;
    }

    return(success);
} // BlpPolDelCategory

// -----
// Function:
//   BlpPolTestCategory
// Inputs:
//   secLabel      The secrecy label to use when testing whether the
//                  category is currently present (INCLUDED) or not
//                  (EXCLUDED).
//   category      The specific category to look for.
// Outputs:
//   status        A boolean value indicating whether the category is
//                  present (TRUE) or not (FALSE).
//   <result>      The success or failure of the operation.  A value of
//                  NO_ERROR indicates a success, while any other value
//                  indicates an error.
// Description:
//   This entry point is used to test whether a particular category is
//   in the set of stored categories in the given BLP secrecy label.
// -----
int BlpPolTestCategory( const BlpLabelType  secLabel,
                        const BlpCatType    category,
                        Boolean             *status)
{
    int success = NO_ERROR;
    *status      = FALSE;

    if (category < MAX_SEC_CATS) {
        *status = secLabel >> (NUM_LEVEL_BITS + category);
    }
    else {
        success = BLPPOL_BADCATEGORY;
    }

    return(success);
} // BlpPolTestCategory
// END OF blppol_getset.c

```

3. blppol_access.c

```
// -----
// File: blppol_access.c
// -----
// Description: This is the implementation file for the following entry
// points of the Bell and LaPadula (BLP) Policy Module:
//     1. BlpPolDominates
//     2. BlpPolRead
//     3. BlpPolWrite
//
// Created: 29-Jul-99 (P. Clark)
//
// Modifications:
// -----

#include <pel/blppol.h>

// -----
// Function:
//     BlpPolDominates
// Inputs:
//     highLabel    The BLP secrecy label to be tested for dominance
//                  against lowLabel.
//     lowLabel     The BLP secrecy label to be tested for dominance
//                  against highLabel.
// Outputs:
//     dominates    A value of TRUE or FALSE, indicating whether the
//                  input highLabel dominates the input lowLabel.
//     <result>     The success or failure of the operation. A value of
//                  NO_ERROR indicates a success, while any other value
//                  indicates an error.
// Description:
//     This entry point compares the first BLP secrecy label with the
//     second secrecy label and communicates whether the first label
//     dominates the second label.
// -----

int BlpPolDominates( const BlpLabelType  highLabel,
                    const BlpLabelType  lowLabel,
                    Boolean               *dominates )
{
    int          success = NO_ERROR;
    int          bit;
    int          done;
    BlpLevelType levelHigh;
    BlpLevelType levelLow;
    Boolean      catHigh;
    Boolean      catLow;

    *dominates = FALSE;
    success    = BlpPolGetLevel( highLabel, &levelHigh );
```

```

    if (success == NO_ERROR) {
        success = BlpPolGetLevel( lowLabel, &levelLow );
    }

    if ((success == NO_ERROR) && (levelHigh >= levelLow)) {
        // make sure that the categories in highLabel are a
        // superset of the categories in lowLabel.

        for (bit=0, done=FALSE; (bit<MAX_SEC_CATS) && (!done);
++bit) {

            success = BlpPolTestCategory(
                highLabel,
                bit,
                &catHigh );

            if (success == NO_ERROR) {
                success = BlpPolTestCategory(
                    lowLabel,
                    bit,
                    &catLow);
            } else {
                done = TRUE;
            }
            if (success == NO_ERROR) {
                if ((catHigh==FALSE) && (catLow==TRUE)) {
                    // high doesn't dominate low
                    done = TRUE;
                }
            }
        }
        if (done == FALSE) {
            *dominates = TRUE;
        }
    }

    return(success);
} // BlpPolDominates

```

```

// -----
// Function:
//   BlpRead
// Inputs:
//   subjectLabel The BLP secrecy label for the subject requesting
//               read access.
//   objectLabel  The BLP secrecy label for the object to be read.
// Outputs:
//   access       A value of ALLOWED or DISALLOWED, indicating whether
//               the read operation requested by the associated
//               subject is allowed for the associated object, or
//               whether it should be disallowed.

```

```

//      <result>      The success or failure of the operation.  A value of
//                      NO_ERROR indicates a success, while any other value
//                      indicates an error.
// Description:
//      The entry point determines whether read access is allowed by the
//      Bell and LaPadula policy, given the input subject and object
//      secrecy labels.
// -----

```

```

int BlpPolRead( const BlpLabelType  subjectLabel,
                const BlpLabelType  objectLabel,
                Boolean              *access)
{
    int      success;
    Boolean   dominates;

    success   = NO_ERROR;
    dominates = FALSE;
    *access   = DISALLOWED;

    success = BlpPolDominates( subjectLabel, objectLabel,
&dominates);

    if ((success == NO_ERROR) && (dominates)) {
        *access = ALLOWED;
    }

    return(success);
} // BlpPolRead

```

```

// -----
// Function:
//      BlpWrite
// Inputs:
//      subjectLabel The BLP secrecy label for the subject requesting
//                      write access.
//      objectLabel  The BLP secrecy label for the object to be modified.
// Outputs:
//      allowed       A boolean value of TRUE or FALSE, indicating whether
//                      the write operation requested by the associated
//                      subject is allowed for the associated object (TRUE),
//                      or whether it should be disallowed (FALSE).
//      <result>      The success or failure of the operation.  A value of
//                      zero indicates a success, while any other value
//                      indicates an error.
// Description:
//      This entry point determines whether a subject may have write
//      access, with respect to the Bell and LaPadula policy, given the
//      input subject and object secrecy labels.  A write operation is
//      only allowed if both labels are valid and equal.
// -----

```

```

int BlpPolWrite( const BlpLabelType  subjectLabel,
                 const BlpLabelType  objectLabel,
                 Boolean              *access)
{

```

```

    int      success;
    Boolean dominates;

    success = NO_ERROR;
    dominates = FALSE;

    success = BlpPolDominates( subjectLabel, objectLabel, &dominates
);

    if ((success == NO_ERROR) && (dominates)) {
        success = BlpPolDominates(
            objectLabel, subjectLabel, &dominates);

        if ((success == NO_ERROR) && (dominates)) {
            *access = ALLOWED;
        }
    }

    return(success);
} // BlpPolRead

// END OF blppol_access.c

```

C. LABEL MANAGER (LBL_MGR)

1. lblmgr.h

```

// -----
// File: lblmgr.h
// -----
// Description: This file contains the external interface to the Label
//              Manager module (LBL_MGR).
//
// Created: 04-Aug-99 (P. Clark)
//
// Modifications:
// -----

#ifndef _LBLMGR_H_
#define _LBLMGR_H_

#include <pel/peltyp.h>
#include <pel/blppol.h>
#include <pel/bibpol.h>

#define MAX_HRL_STR      256
#define DELIMETER        ':'

typedef Bits64 PolicyLabelType;

```

```

extern Boolean LblMgrIsValid(  const PolicyLabelType binaryLabel );

extern void    LblMgrInitLabel(      PolicyLabelType *binaryLabel );

extern int     LblMgrBinToStr( const PolicyLabelType  binaryLabel,
                               char                    stringLabel[] );

extern int     LblMgrStrToBin( const char              stringLabel[],
                               PolicyLabelType *binaryLabel );

extern int     LblMgrGetBlp(  const PolicyLabelType policyLabel,
                               BlpLabelType          *secrecyLabel );

extern int     LblMgrGetBiba( const PolicyLabelType policyLabel,
                               BibLabelType          *integrityLabel );

extern int     LblMgrSetBlp(  const BlpLabelType      secrecyLabel,
                               PolicyLabelType *policyLabel );

extern int     LblMgrSetBiba( const BibLabelType      integrityLabel,
                               PolicyLabelType *policyLabel);

// Error codes returned by this module.
//
#define LBLMGR_ERRBASE      1800
#define LBLMGR_BADSTRING    LBLMGR_ERRBASE
#define LBLMGR_BADLABEL     LBLMGR_ERRBASE+1
#define LBLMGR_UNEXPECTED   LBLMGR_ERRBASE+2

#endif

// EOF for lblmgr.h

```

2. lblmgr_getset.c

```

// -----
// File: lblmgr_getset.c
// -----
// Description: This is the implementation file for the following
// external entry points for the Label Manager module (LBL_MGR):
//     LblMgrIsValid()
//     LblMgrInitLabel()
//     LblMgrGetBlp()
//     LblMgrSetBlp()
//     LblMgrGetBiba()
//     LblMgrSetBiba()
//
// Created: 04-Aug-99 (P. Clark)
//
// Modifications:
// -----

```

```
#include <pel/peltyp.h>
#include <pel/blppol.h>
#include <pel/bibpol.h>
#include <pel/lblmgr.h>
```

```
#define MIN_VERSION          1
#define MAX_VERSION          15
#define CUR_VERSION          1
#define NUM_VERSION_BITS     4
#define SECRECY_MASK         0x0ff0
#define VERSION_MASK         0x000f
```

```
// Where L is of type PolicyLabelType
```

```
//
```

```
#define GET_SECRECY(L)  ((L & SECRECY_MASK) >> NUM_VERSION_BITS)
```

```
// -----
// Function:
//   LblMgrInitLabel
// Inputs:
//   <none>
// Outputs:
//   binaryLabel      An initialized Policy Label.
// Description:
//   This entry point is used to initialize a Policy Label in two
//   ways:
//   1) properly set the version field to avoid LBLMGR_BADLABEL
//   errors;
//   2) getting initialized secrecy and integrity labels and storing
//   them in the Policy Label.
// -----
```

```
void LblMgrInitLabel( PolicyLabelType *binaryLabel )
```

```
{
```

```
    // NOTE THAT INTEGRITY LABELS ARE NOT INITIALIZED YET
```

```
    BlpLabelType tmpSecrecy;
```

```
    *binaryLabel = CUR_VERSION;
```

```
    BlpPolInitLabel( &tmpSecrecy );
```

```
    LblMgrSetBlp( tmpSecrecy, binaryLabel );
```

```
} // LblMgrInitLabel()
```

```
// -----
// Function:
//   LblMgrIsValid
// Inputs:
//   binaryLabel      The binary Policy Label to test for validity.
```

```

// Outputs:
//   <result>          A value of VALID is returned if the input
//                      binaryLabel is valid.  Otherwise, a value of
//                      INVALID is returned.
// Description:
//   This entry point is used to test whether a given label is valid
//   or not.
// -----
Boolean LblMgrIsValid( const PolicyLabelType binaryLabel )
{
    unsigned int    lblVersion;
    Boolean         result;

    result = VALID;
    lblVersion = (unsigned int)(binaryLabel & VERSION_MASK);

    if ((lblVersion < MIN_VERSION) || (lblVersion > MAX_VERSION)) {
        result = INVALID;
    }

    return(result);
} // LblMgrIsValid()

```

```

// -----
// Function:
//   LblMgrGetBlp
// Inputs:
//   policyLabel  A binary Policy Label.
// Outputs:
//   secrecyLabel The BLP secrecy portion of the input Policy Label.
//   <result>     The success or failure of the operation.  A value
//               of NO_ERROR indicates a success, while any other
//               value indicates an error.
// Description:
//   This entry point is used to copy the BLP portion of a Policy
//   Label and return it to the caller.
// -----
int LblMgrGetBlp( const PolicyLabelType policyLabel,
                  BlpLabelType *secrecyLabel )
{
    int          success;
    unsigned int  lblVersion;
    PolicyLabelType tmp;

    success = NO_ERROR;
    lblVersion = (unsigned int)(policyLabel & VERSION_MASK);

    if ((lblVersion < MIN_VERSION) || (lblVersion > MAX_VERSION)) {
        success = LBLMGR_BADLABEL;
    }
    else {
        // Perform processing based on the version number
        // of the Policy Label
        switch (lblVersion) {
            case 1: {

```

```

        *secrecyLabel = GET_SECRECY(policyLabel);
        break;
    }
    default: {
        // This should never happen
        success = LBLMGR_UNEXPECTED;
        break;
    }
}

return(success);
} // LblMgrGetBlp()

// -----
// Function:
//   LblMgrSetBlp
// Inputs:
//   secrecyLabel  The secrecy label to copy into the Policy Label.
// Outputs:
//   policyLabel   A Policy Label with the input secrecy label
//                 properly installed.
//   <result>      The success or failure of the operation.  A value
//                 of NO_ERROR indicates a success, while any other
//                 value indicates an error.
// Description:
//   This entry point is used to set the BLP secrecy portion of a
//   Policy Label.
// -----
int LblMgrSetBlp( const BlpLabelType    secrecyLabel,
                  PolicyLabelType *policyLabel )
{
    int          success;
    unsigned int  lblVersion;
    PolicyLabelType tmpSec;
    PolicyLabelType tmpPol;

    success      = NO_ERROR;
    lblVersion = (unsigned int) ((*policyLabel) & VERSION_MASK);

    if ((lblVersion < MIN_VERSION) || (lblVersion > MAX_VERSION)) {
        success = LBLMGR_BADLABEL;
    }
    else {
        // Perform processing based on the version number
        // of the Policy Label
        switch (lblVersion) {
            case 1: {
                // 1st get rid of old secrecy label
                tmpPol = (*policyLabel) & (~SECRECY_MASK);

                // then copy in the new stuff
                tmpSec = secrecyLabel;
                tmpPol = tmpPol | (tmpSec << NUM_VERSION_BITS);
                *policyLabel = tmpPol;
            }
        }
    }
}

```

```

        break;
    }
    default: {
        // This should never happen
        success = LBLMGR_UNEXPECTED;
        break;
    }
}

return(success);
} // LblMgrSetBlp()

```

// EOF lblmgr_getset.c

D. META-POLICY MANAGER (POL_MGR)

1. polmgr.h

```

// -----
// File: polmgr.h
// -----
// Description: This file contains the external interface to the Meta-
//              Policy Manager (POL_MGR).
//
// Created: 16-Aug-99 (P. Clark)
//
// Modifications:
// -----

#ifndef _POLMGR_H_
#define _POLMGR_H_

#include <pel/lblmgr.h>

extern int PolMgrDominates( const PolicyLabelType highLabel,
                           const PolicyLabelType lowLabel,
                           Boolean *dominates );
extern int PolMgrRead(     const PolicyLabelType subjectLabel,
                           const PolicyLabelType objectLabel,
                           Boolean *allowed );
extern int PolMgrWrite(    const PolicyLabelType subjectLabel,
                           const PolicyLabelType objectLabel,
                           Boolean *allowed );

#endif

// EOF for polmgr.h

```

2. polmgr.c

```
// -----
// File: polmgr.c
// -----
// Description: This is the implementation file for the Meta-Policy
// Manager (POL_MGR). The following external entry points are
// implemented in this file:
//     1. PolMgrDominates()
//     2. PolMgrRead()
//     3. PolMgrWrite()
//
// Created: 16-Aug-99 (P. Clark)
//
// Modifications:
// -----

#include <pel/peltyp.h>
#include <pel/blppol.h>
#include <pel/lblmgr.h>
#include <pel/polmgr.h>

// -----
// Function:
//     PolMgrDominates
// Inputs:
//     highLabel    The Policy Label to use when testing for dominance.
//     lowLabel     The Policy Label to use as the dominated label.
// Outputs:
//     dominates    TRUE indicates that highLabel dominates lowLabel.
//                 FALSE indicates that highLabel does not dominate
//                 lowLabel. It cannot be assumed that lowLabel
//                 dominates highLabel if FALSE is returned.
//     <result>     The success of failure of the operation. A value of
//                 NO_ERROR indicates a success, while any other value
//                 indicates an error.
// Description:
//     This entry point is used to test whether one Policy Label
//     dominates another.
// -----
int PolMgrDominates( const PolicyLabelType  highLabel,
                    const PolicyLabelType  lowLabel,
                    Boolean                 *dominates )
{
    // *****
    // NOTE THAT THE BIBA POLICY IS NOT YET BEING CALLED HERE
    // *****

    int          success;
    BlpLabelType blpLow;
    BlpLabelType blpHigh;

    success      = NO_ERROR;
```

```

    *dominates = FALSE;

    success = LblMgrGetBlp( lowLabel, &blpLow );

    if (success == NO_ERROR) {
        success = LblMgrGetBlp( highLabel, &blpHigh );
    }

    if (success == NO_ERROR) {
        success = BlpPolDominates( blpHigh, blpLow, dominates );
    }

    return(success);
} // PolMgrDominates()


// -----
// Function:
//   PolMgrRead
// Inputs:
//   subjectLabel The binary Policy Label representing the level of
//               the subject that wants to read the object.
//   objectLabel  The binary Policy Label representing the level of
//               the object to be read.
// Outputs:
//   access       A value of ALLOWED is returned if the subject
//               associated with the input subjectLabel is allowed to
//               read the object associated with the input
//               objectLabel. Otherwise a value of DISALLOWED is
//               returned.
//   <result>     The success or failure of the operation. A value of
//               NO_ERROR indicates a success, while any other value
//               indicates an error.
// Description:
//   This entry point is used to determine whether the associated
//   subject should be allowed to read the associated object, given
//   the subject and object Policy Labels. The operation is allowed
//   if the input subjectLabel dominates the input objectLabel.
// -----
int PolMgrRead( const PolicyLabelType subjectLabel,
                const PolicyLabelType objectLabel,
                Boolean *access)
{
    int success;
    Boolean result;

    success = NO_ERROR;
    result = FALSE;
    *access = DISALLOWED;

    success = PolMgrDominates( subjectLabel, objectLabel, &result );

    if ((success == NO_ERROR) && (result == TRUE)) {
        *access = ALLOWED;
    }
}

```

```

        return(success);
    } // PolMgrRead()

```

```

// -----
// Function:
//   PolMgrWrite
// Inputs:
//   subjectLabel The binary Policy Label representing the level of
//               the subject that wants to write the object.
//   objectLabel  The binary Policy Label representing the level of
//               the object to be modified.
// Outputs:
//   access       A value of ALLOWED is returned if the subject
//               associated with the input subjectLabel is allowed to
//               write the object associated with the input
//               objectLabel.
//               Otherwise, a value of DISALLOWED is returned.
//   <result>     The success or failure of the operation. A value of
//               NO_ERROR indicates a success, while any other value
//               indicates an error.
// Description:
//   This entry point is used to determine whether the associated
//   subject should be allowed to write the associated object, given
//   the subject and object Policy Labels. The only way that write
//   permission will be allowed is if the subject and object labels
//   are equal, viz., the two labels dominate each other.
// -----

```

```

int PolMgrWrite( const PolicyLabelType subjectLabel,
                 const PolicyLabelType objectLabel,
                 Boolean *access)
{
    int success;
    Boolean dominates;

    success = NO_ERROR;
    *access = DISALLOWED;
    dominates = FALSE;

    success = PolMgrDominates( subjectLabel, objectLabel, &dominates
);

    if ((success == NO_ERROR) && (dominates)) {
        success = PolMgrDominates(
            objectLabel, subjectLabel, &dominates);

        if ((success == NO_ERROR) && (dominates)) {
            *access = ALLOWED;
        }
    }

    return(success);
} // PolMgrWrite()

// END OF polmgr.c

```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1. *Critical Foundations, Protecting America's Infrastructures*, The Report of the President's Commission on Critical Infrastructure Protection, October, 1997.
2. *The Clinton Administration's Policy on Critical Infrastructure Protection: Presidential Decision Directive 63*, White Paper, May 22, 1998, <http://131.84.1.84/6263summary.html>
3. Gasser, Morrie, *Building a Secure Computer System*. New York: Van Nostrand Reinhold, 1988.
4. Pfleeger, Charles P., *Security in Computing*. Second edition, Upper Saddle River: Prentice Hall, Inc., 1997.
5. Bishop, Matt, *History of Computer Security Project, Early Papers in the Field of Computer Security, CD-ROM #1*, University of California at Davis, October 1998.
6. Department of Defense, *Security Requirements for Automatic Data Processing (ADP) Systems*, Directive 5200.28, April 1978.
7. Department of Defense, *Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD, December 1985.
8. Bell, D.E., La Padula, L.J., *Secure Computer System: Unified Exposition and Multics Interpretation*, Electronic Systems Division, Air Force Systems Command, United States Air Force, Hanscom Air Force Base, Report MTR-2997 Rev. 1, March, 1976.
9. LaPadula, Leonard J., Bell D. Elliott, *MITRE Technical Report 2547, Volume II*, Journal of Computer Security, Volume 4, Nos. 2, 3, pp. 239-263, 1996.
10. Higgins, John C., "Information Security as a Topic in Undergraduate Education of Computer Scientists," *Proceedings of the 12th National Computer Security Conference*, National Institute of Standards and Technology / National Computer Security Center, pp. 553-557, October 1989.
11. <http://agn-www.informatik.uni-hamburg.de/people/1ott/rsbac>
12. Beck, Michael, Bohme, Harald, Dziadzka, Mirko, Kunitz, Ulrich, Magnus, Robert, Verworner, Dirk, *Linux Kernel Internals*. Second edition, New York: Addison-Wesley, 1998.
13. Irvine, Cynthia E., "A Multilevel File System for High Assurance", *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, Institute of Electrical and Electronics Engineers, Inc. [IEEE], pp. 78-87, 1995.

14. Kramer, Steven, "Linus IV - An Experiment in Computer Security", *Proceedings of the 1984 Symposium on Security and Privacy*, Institute of Electrical and Electronics Engineers, Inc. [IEEE], pp. 24-31, 1984.
15. *Final Evaluation Report, Trusted Information Systems, Inc., Trusted XENIX version 4.0*, National Computer Security Center, Report CSC-EPL-92/001.A, January, 1994.
16. *Final Evaluation Report, Silicon Graphics Computer Systems, Inc., Trusted IRIX/B*, National Computer Security Center, Report CSC-EPL-95/001, February, 1995.
17. *Final Evaluation Report, Wang Federal Incorporated, XTS-300*, National Computer Security Center, Report CSC-EPL-92/003.B, July 1995.
18. Draft Security Interface for the Portable Operating System Interface for Computer Environments, Technical Committee on Operating Systems and Operational Environments of the IEEE Computer Society, P1003.6 / D11, May 1991.
19. Saltzer, Jerome H., Schroeder, Michael D., "The Protection of Information in Computer Systems", *Proceedings of the IEEE*, Volume 63, pp. 1278-1308, 1974.
20. Summers, Rita C., *Secure Computing, Threats and Safeguards*. New York: McGraw-Hill, 1997.
21. Levin, Tim, Padilla, Steven J., Irvine, Cynthia E., "A Formal Model for Unix Setuid", *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, Institute of Electrical and Electronics Engineers, Inc. [IEEE], pp. 73-83, 1989.
22. Ko, Calvin, Rushchitzka, Manfred, Levitt, Karl, "Execution of Security-Critical Programs in Distributed Systems: A Specification-based Approach", *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, Institute of Electrical and Electronics Engineers, Inc. [IEEE], pp. 175-187, 1997.
23. Saydjari, O. Sami, Beckman Joseph M., Leaman, Jeffrey R., "LOCK Trek: Navigating Uncharted Space", *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, Institute of Electrical and Electronics Engineers, Inc. [IEEE], pp. 167-175, 1989.
24. Ilgun, Koral, "USTAT: A Real-time Intrusion Detection System for UNIX", *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, Institute of Electrical and Electronics Engineers, Inc. [IEEE], pp. 16-28, 1993.
25. *A Guide to Understanding Covert Channel Analysis of Trusted Systems*, National Computer Security Center, NCSC-TG-030, Version 1, November 1993.

26. Lampson, Butler W., "A Note of the Confinement Problem", *Communications of the ACM*, Volume 16, pp. 613-615, 1973.
27. Card, Rémy, Dumas, Éric, Mével, Franck, *The Linux Kernel Book*. West Sussex: John Wiley and Sons, 1998.
28. Rusling, David A. *The Linux Kernel*, <http://metalab.unc.edu/mdw/LDP/tlk/tlk.html>

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
 8725 John J. Kingman Rd., Ste 0944
 Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
 Naval Postgraduate School
 411 Dyer Rd.
 Monterey, CA 93943-5101

3. Chairman, Code CS 1
 Computer Science Department
 Naval Postgraduate School
 Monterey, CA 93943-5000

4. Dr. Cynthia E. Irvine 2
 Computer Science Department Code CS/Ic
 Naval Postgraduate School
 Monterey, CA 93943-5000

5. Dr. Dennis Volpano 1
 Computer Science Department Code CS/Vd
 Naval Postgraduate School
 Monterey, CA 93943-5000

6. Mr. James P. Anderson 1
 James P. Anderson Company
 Box 42
 Fort Washington, PA 19034

7. Paul Pittelli 1
 National Security Agency
 Research and Development Building
 R2
 9800 Savage Road
 Fort Meade, MD 20755-6000

8. CAPT Dan Galik 1
 Space and Naval Warfare Systems Command
 PMW 161
 Building OT-1, Room 1024
 4301 Pacific Highway
 San Diego, CA 92210-3127

9. Commander, Naval Security Group Command 1
Naval Security Group Headquarters
9800 Savage Road
Suite 6585
Fort Meade, MD 20755-6585
10. Mr. George Bieber 1
Defense Information Systems Agency
Center for Information Systems Security
5113 Leesburg Pike, Suite 400
Falls Church, VA 22041-3230
11. Louise Davidson 1
N643
Presidential Tower 1
2511 South Jefferson Davis Highway
Arlington, VA 22202
12. Lt. Col. Timothy Fong 1
Defense Information Systems Agency
5600 Columbia Pike
Falls Church, VA 22041
13. Mr. Paul Clark 1
Computer Science Department Code CS/Cp
Naval Postgraduate School
411 Dyer Rd.
Monterey, CA 93943-5000